

PDP-11
Assembler Language
Programming and
Machine
Organization

Michael Singer

INSTRUCTIONS

DD = 6-bit code for destination

R = 3-bit code for register

SS = 6-bit code for source

x = 0 for word instructions, 1 for byte instructions

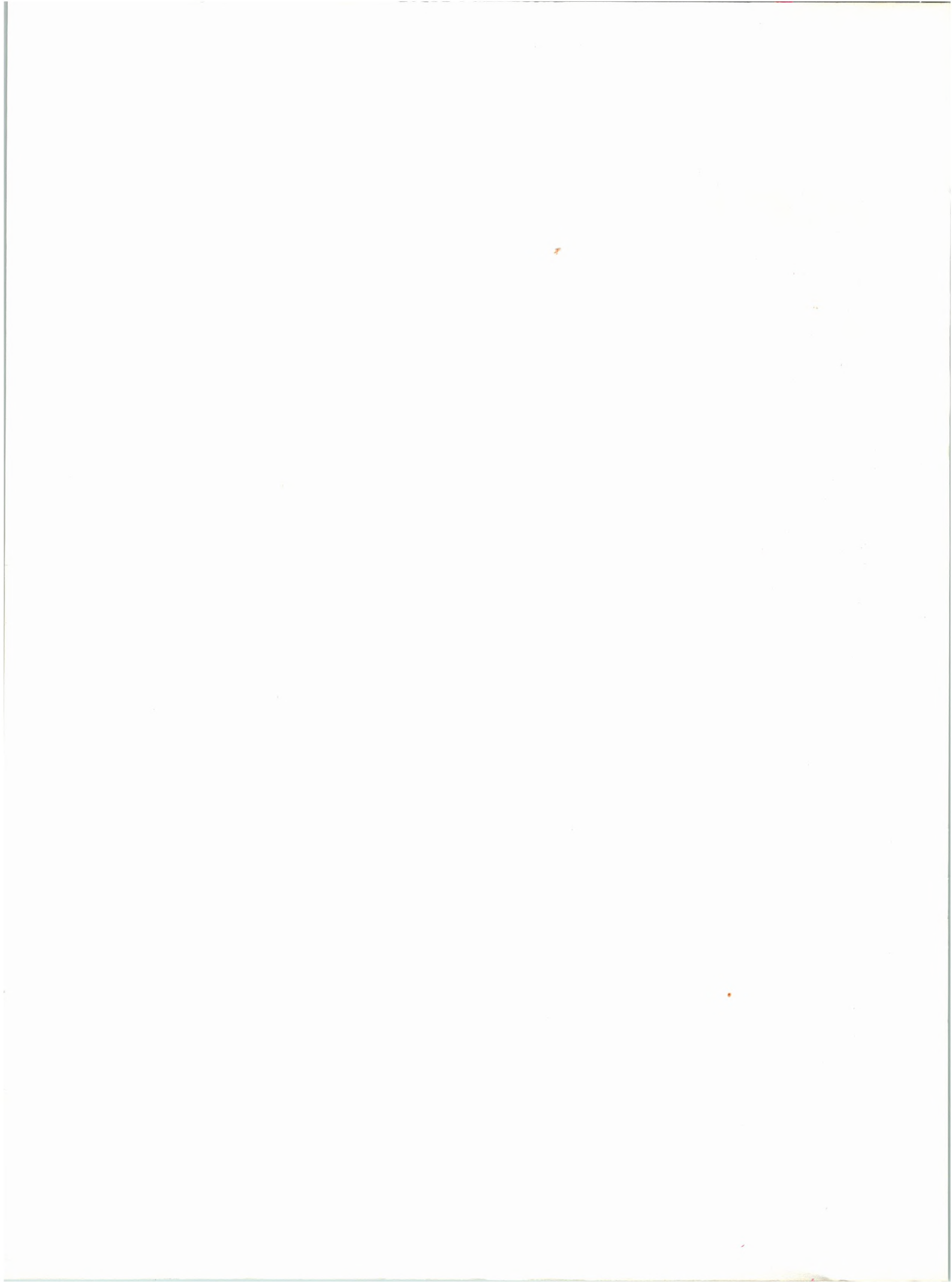
	<i>Mnemonic</i>	<i>Code</i>	<i>Description</i>	<i>Condition Codes Affected</i>
OPERAND	CLR(B)	x050DD	clear dst	N, V, C clr; Z set
	DEC(B)	x053DD	subtract 1 from dst	N, Z, V set/clr by result
	INC(B)	x052DD	add 1 to dst	N, Z, V set/clr by result
	NEG(B)	x054DD	negate dst	N, Z, V set/clr by result
	TST(B)	x057DD	set condition codes	C clr if result = 0, else set N, Z set/clr by dst contents V, C cleared
	COM(B)	x051DD	complement dst	N, Z set/clr by result, V clr, C set
SINGLE	ASR(B)	x062DD	shift dst 1 place right replicate high-order bit	N, Z set/clr by result C ← old low-order bit of dst V ← exclusive OR of N, C bits
	ASL(B)	x063DD	shift dst 1 place left put 0 in low-order bit	N, Z, set/clr by result C ← old high-order bit of dst V ← exclusive OR of N, C bits
	ADC(B)	x055DD	add C bit to dst	all set/clr by result
	SBC(B)	x056DD	subtract C bit from dst	all set/clr by result
	SXT	0067DD	all dst bits to value of N bit	Z set if N bit clr, V clr
	ROR(B)	x060DD	rotate dst right 1 bit, via C bit	N, Z set/clr by result C ← old low-order bit of dst V ← exclusive OR of N, C bits
	ROL(B)	x061DD	rotate dst left 1 bit, via C bit	N, Z set/clr by result C ← old high-order bit of dst V ← exclusive OR of N, C bits
	SWAB	0003DD	swap dst bytes	N, Z set/clr by old dst value V, C clr

BRANCHES: PC ← PC + (2 × offset). Coded as code + offset. Condition codes unaffected.

	<i>Mnemonic</i>	<i>Code</i>	<i>Condition</i>		<i>Mnemonic</i>	<i>Code</i>	<i>Condition</i>
BRANCHES	BR	000400	always		BNE	001000	Z = 0
	BEQ	001400	Z = 1		BPL	100000	N = 0
	BMI	100400	N = 1		BCC	103000	C = 0
	BCS	103400	C = 1		BVC	102000	V = 0
	BVS	102400	V = 1		BGE	002000	N ≠ V = 0
	BLT	002400	N ≠ V = 1		BGT	003000	Z ∨ (N ≠ V) = 0
	BLE	003400	Z ∨ (N ≠ V) = 1		BHI	101000	C = 0 and Z = 0
	BLOS	101400	C ∨ Z = 1				

PDP-11

Assembler Language
Programming
and Machine
Organization



PDP-11

Assembler Language Programming and Machine Organization

MICHAEL SINGER
Stanford University

JOHN WILEY & SONS
NEW YORK CHICHESTER BRISBANE TORONTO

Copyright © 1980, by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons.

Library of Congress Cataloging in Publication Data:

Singer, Michael, 1942-

PDP-11 assembler language programming and machine organization.

Includes indexes.

1. PDP-11 (Computer)—Programming. 2. Assembler language (Computer program language) I. Title.

QA76.8.P2S56 001.64 80-10103

ISBN 0-471-04905-0

Printed in the United States of America

10 9 8 7 6 5 4 3 2

Preface

Today the small computer is an integral part of commercial and scientific establishments throughout the world. It performs as an aid to profitability, a relief from drudgery and, all too often, a source of frustration. This book will help the user to improve matters on the first two counts and to mitigate on the last.

This book is devoted exclusively to the PDP-11 computer, manufactured by the Digital Equipment Corporation (DEC), and one of the most popular and flexible small machines available. All computers share a certain fundamental electronic and conceptual structure, and much of what we say in this book would apply equally well, with relatively minor changes, to other machines. We did not, however, set out to place on the market yet another general text on computer organization. Computer programming is a practical art that is to be bolstered by theoretical wisdom but developed as a skill by continual practice. Since, at the computer terminal, the beginner is a good deal more aware of the practical differences than the structural similarities between various machines, we feel that an introduction of this kind should focus, and remain focused, on a particular computer.

It is inherent to our approach that the reader will be writing complete programs, although naturally rather trivial ones, at the earliest possible stage. Thus, access to a PDP-11 installation is helpful from the outset. There are no other prerequisites; in particular, we assume no prior experience with or theoretical knowledge of computers. This book may be read independently of other studies by the beginning or advanced PDP-11 user. It will also complement a typical first or second course in computer programming.

It is no more possible to understand the workings of a computer than of a society while remaining ignorant of the usages and subtleties of its language. The language of a computer is its own so-called machine code; it can "understand" no other and will only accept statements in a higher-level language such as BASIC, COBOL, FORTRAN, or PASCAL if these are translated for it. The translation program, called a *compiler*, may itself have been written in a higher-level language. But at some stage the computer must be told what the higher-level statements "mean" in terms of its own machine code. Thus, if we are to enjoy the fullest range of communication with the PDP-11, we must understand its machine code. Even PDP-11 assembler language, which is virtually a direct restatement of machine code instructions in a readable format, forms a barrier between user and machine. The barrier is, however, of slight proportions, and the assembler language programmer has all the facilities of the PDP-11 available; although an understanding of the mechanics of machine code is necessary to take full advantage of them.

Even a person who does not normally program in assembler language will benefit from a closer awareness of how the computer works. It is, however, very hard to be certain that one will never need assembler language. A higher-level language designed to operate uniformly on all machines may well not reflect the special facilities available on any one machine, and that may be needed for a particular task.

Assembler language may also be more efficient in certain cases. On a small machine, the compiler for a higher-level language cannot, because of size restrictions, incorporate the refinements of optimization necessary for production of the most efficient code. Thus execution time, and hence cost, will be increased. This argument is unfashionable in certain circles, where it is customary to point out that an inefficient method of approach, or *algorithm*, is more wasteful than inefficiencies in the details of coding, and that the greater value of human than machine time may justify even the use of inefficient

algorithms. We would agree that human time is more important than machine time, but human time may equally be absorbed in waiting for another human being's inefficient program to finish running. More fundamentally, vital human resources are squandered if four computers are used wastefully instead of three efficiently. Nor would we disagree that a good algorithm inefficiently coded is usually better than a poor algorithm on which the most skillful programmer has poured out energy. This, however, is not the precise issue. A proper cost/benefit analysis should be made in which all these competing values are weighed. Clearly, a realistic analysis can only be made by a programmer fully trained in all aspects of the art; improving the efficiency of coding is never worth the effort until one knows how to do it.

Although this is not a text on the design of algorithms (indeed, the term "algorithm" is not employed outside this preface), we nevertheless always think in terms of a properly structured approach to programming. Thus, we have avoided the temptation of building a book around displays of our own favorite, and lengthy, programs. Instead, this text abounds with examples of how to encode a great variety of small individual tasks. Chapter 3, the heart of the book, is then devoted to the PDP-11's facilities for building complete programs out of these individual blocks, or *modules*. As much as possible, we avoid considering logical flow in the abstract, but we take great care that this most important of programming concepts permeates the entire discussion.

Chapter 1 introduces the computer system. It includes the operating system insofar as this is necessary for the goal of the chapter, which is to get a simple program to work. Our aim of allowing practice gently to lead theory requires that students be able to experiment, and leads to the anomaly of introducing monitor-controlled output at an early stage. Of course, a full explanation of the workings of monitor calls is considerably postponed. Users of a PDP-11 without an operating system can read Section 4.1 along with Chapter 1 to get started; that section has been written with little reliance on earlier chapters to make this possible.

Chapter 2 offers a broad range of assembler language instructions in problem-solving contexts. We introduce only what can be used forthwith, so many instructions do not appear until later in the book. However, the book as a whole covers the entire range of PDP-11 assembler language instructions.

Our attitude to operating systems remains ambivalent throughout the book. To a purist, these are mere programs and have nothing to do with the machine itself. On the other hand, they include many extraordinarily useful programs (such as assemblers and editors) and are so closely coordinated with the machine as to be virtually a part of its total design. Many aspects of PDP-11 usage are therefore considered both in the context of an operating system and in its absence. This is particularly true in Chapter 4, where we discuss storage device input and output in both settings. That chapter also deals with terminal input/output, interrupts, and memory management.

There are two appendices. Appendix A treats ODT, the debugging aid of PDP-11 operating systems. It is designed to be read in parallel with the body of the text; a start should be made on it when studying Chapter 2.

Appendix B is an introduction to multiple precision integer arithmetic and floating point arithmetic on the PDP-11. Multiple precision is a frequent requirement on minicomputers with small word sizes, and our illustrative routines should prove useful.

The text contains collections of exercises, at least some of which should always be done before reading on. Most of the exercises are straightforward tests of understanding, although the time they require varies greatly. The symbol * marks a few problems of somewhat greater difficulty.

Many colleagues have provided encouragement, advice, and nuggets of esoteric information; these particularly include John Benedict of Ross Systems Inc., Steve Kallis of DEC, and Ted Panofsky of the Artificial Intelligence Laboratory, Stanford University. The staff of John Wiley & Sons have provided support and encouragement throughout.

MICHAEL SINGER

Contents

1	PRELIMINARIES	1
1.1	THE COMPUTER SYSTEM	1
1.2	THE EDITOR	3
	Timesharing Systems 4, Opening a File 4, Writing a File 5, Editing a File 6, Character-Oriented Commands 8, Page Structure 9, Secondary Buffers 10	
1.3	STORING AND RETRIEVING INFORMATION	10
	Memory 11, Binary and Octal Notations 11, ASCII Code 13, Addressing 15, The Central Processing Unit 16	
1.4	RUNNING A PROGRAM	18
	Monitor-Controlled Output 19, The Assembler 19, Start Address 20, Labels 20, Types of Instruction 21, The Assembly Process 21, The System Macros Library 22, Listing File 22, The Linker 24	
2	FUNDAMENTALS	25
2.1	THE REGISTERS	25
	Monitor-Controlled Input 26, Numerical Output 28, Numerical Input 31, Comments 32	
2.2	BRANCH INSTRUCTIONS	32
	Numerical Input 34, Condition codes 37, Counting Data Items 38	
2.3	MEMORY	39
	Indexing 41, Debugging 43, Automatic Pointer Stepping 45, Sorting 46	
2.4	WORD FORMAT	48
	Double-Operand Instructions 51, Program Counter Addressing 54, Relative Mode 55, Negative Numbers 56	

3 PROGRAM STRUCTURE 58

- 3.1 SUBROUTINES 58
 - A Text-Editing Program 59, Nesting Subroutines 61, Linkage Register 62, ASCII Text 65, Local Symbols 68
- 3.2 STACKS 69
 - The Stack Pointer 69, Evaluating Arithmetical Expressions 70, Recursive Subroutines 72, The System Stack 74, Coroutines 76, Location Counter 77
- 3.3 CONTROL TRANSFER DECISIONS 79
 - Logical Instructions 79, Overflow 82, Carry 84, Setting the Condition Codes 85, Byte Instructions 86, Shift and Rotate Instructions 87, Encoding Branch Instructions 89
- 3.4 MODULAR PROGRAMMING 90
 - Passing Parameters 90, Parameters in Higher-Level Languages 92, Traps 94, A Trap Servicing Routine 95, Absolute Addressing 97, Program-Generated Traps 99, Patching 100, Debugging Aids 100
- 3.5 STRUCTURED ASSEMBLY 101
 - Macros 101, Macro Arguments 102, Macro Labels 103, Radix Control 106, Conditional Assembly 106, Expression Value Conditionals 108, Branched Conditionals 109, Forward References 110

4 PERIPHERAL DEVICES 112

- 4.1 TERMINAL AND CONSOLE I/O 112
 - The Console Switches 113, Sequential Character Output 114, Character Input 115, Bootstraps 116
- 4.2 INTERRUPTS 118
 - The UNIBUS 119, Enabling Interrupts 120, Terminal Printer Interrupts 122, Priority 124, T Bit 126
- 4.3 STORAGE DEVICES: MONITOR CONTROL 127
 - Monitor Calls 127, Radix-50 Code 128, Device Handlers 129, Opening a File 130, Writing a File 132, Repeat Directives 133, I/O Buffering 134, Reading a File 135, Multiple Buffering 135

- 4.4 STORAGE DEVICES: USER CONTROL 136
Device Registers 137, Formatting 139, Writing and
Reading the Disk 139, Address Relocation 141,
Interrupts 141, Drive and Controller Interrupts 141,
Error Conditions 142
- 4.5 MEMORY MANAGEMENT 143
Memory Pages 144, Enabling Memory Management 145,
Creation of a Physical Address 146, Page Length and
Access 147, Protection Violations 148, Processor
Modes 149, Communication between Spaces 151,
System Stacks 151

A ODT 153

Running ODT 153, Global Symbols 153, Program Examination
and Correction 154, Program Execution 158

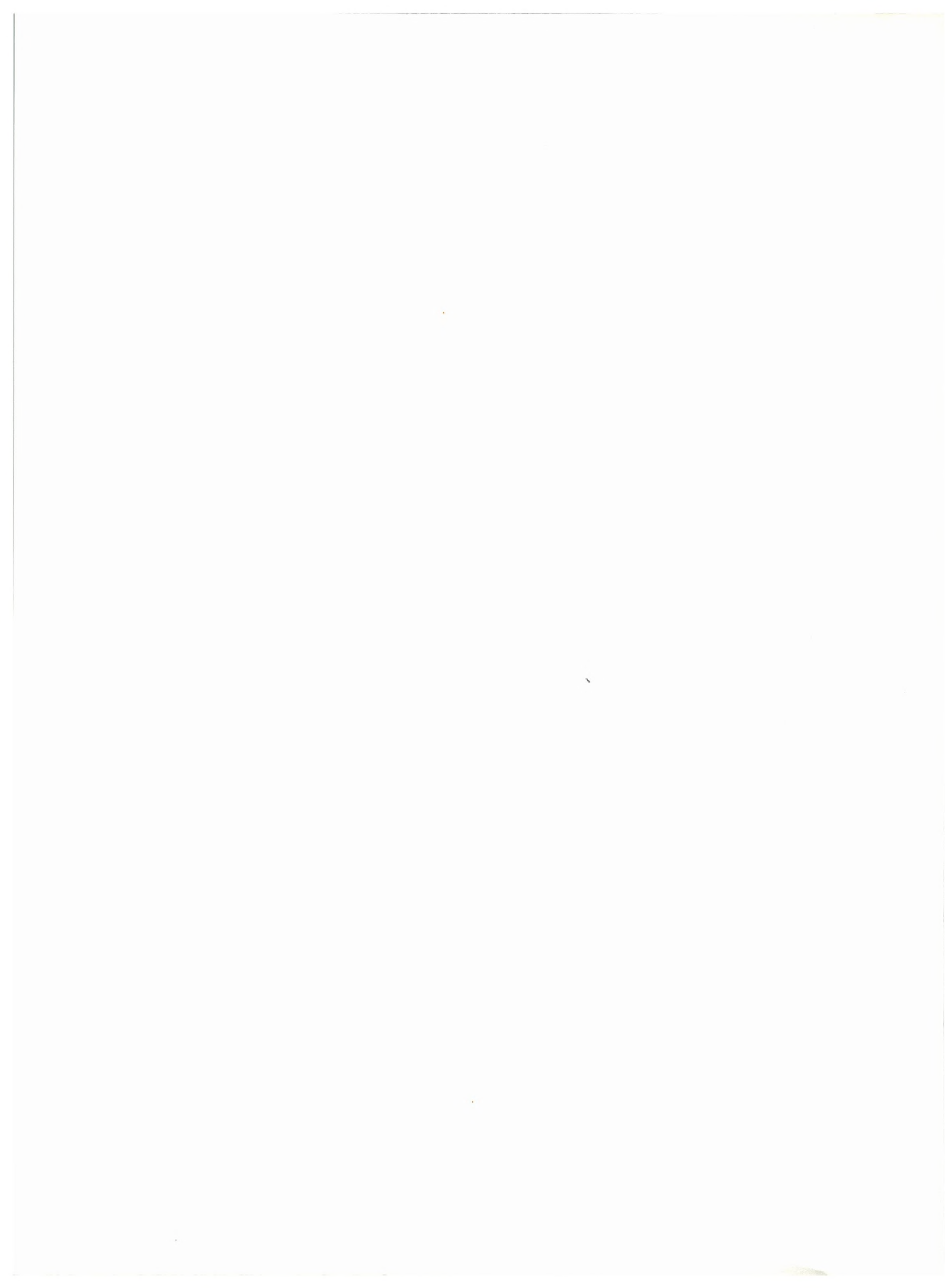
B ARITHMETIC 161

MUL and DIV 161, Multiple Precision Integer Arithmetic 162,
Floating Point Arithmetic 165

ASCII CODES 169

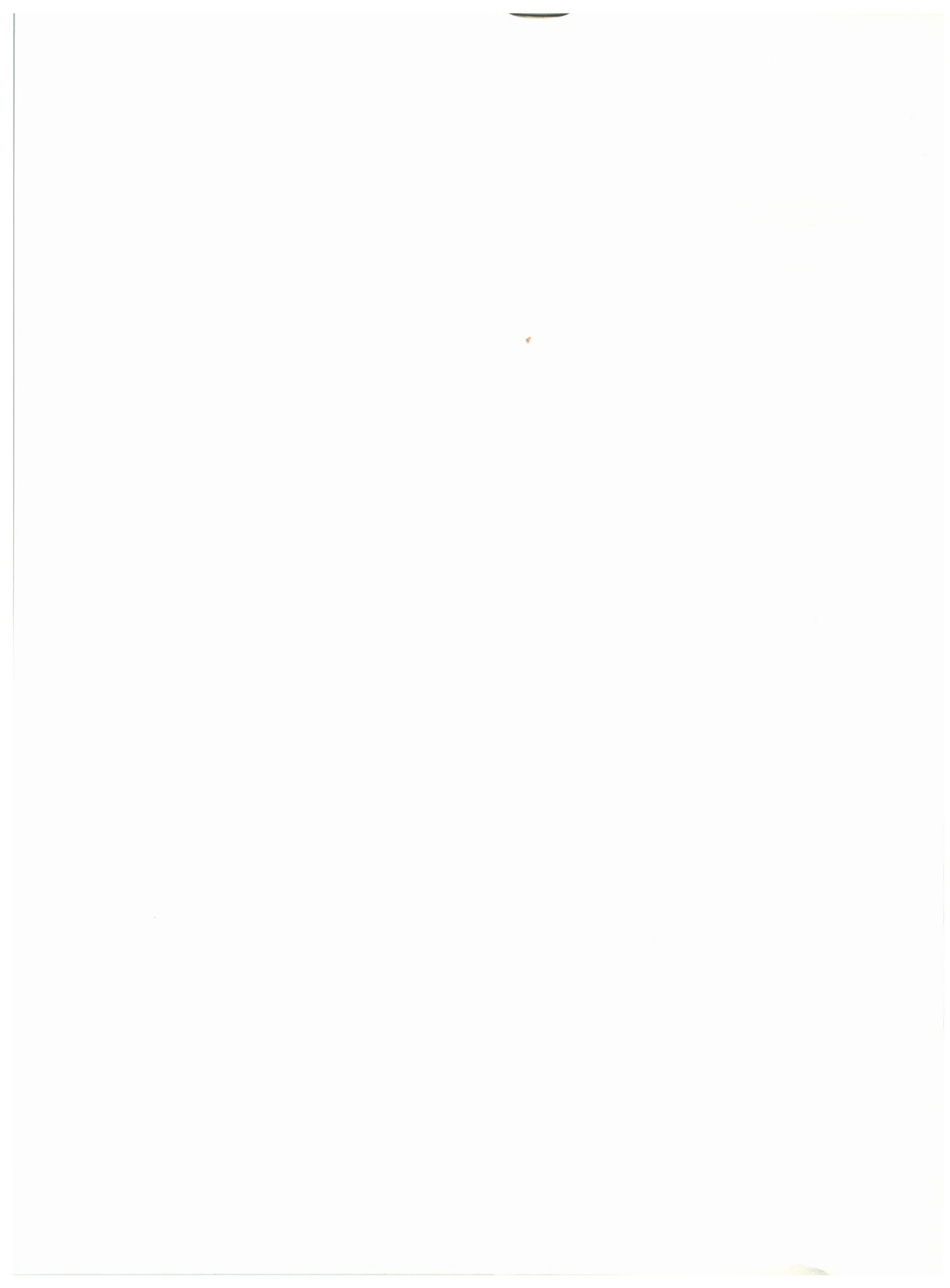
INDEX OF MACRO-11 INSTRUCTIONS 171

SUBJECT INDEX 175



PDP-11

Assembler Language
Programming
and Machine
Organization



1 Preliminaries

1.1 THE COMPUTER SYSTEM

It is most unlikely that the beginning PDP-11 programmer will be faced with a PDP-11 computer in its raw, unembellished state. Normally there will be a *terminal*, through which communication with the machine takes place. There will be devices, usually in the shape of *disks* or *tapes*, on which information can be permanently stored. There are several different models of PDP-11 computer, and the larger ones can serve many users, each at his or her own terminal (a *timesharing* environment); in this case the storage devices will probably be out of sight in the computer room.

Physical devices (*hardware*) such as terminals and disks are known as *peripherals*. The peripherals available for use with PDP-11 computers range in level of sophistication from interfaces that enable them to communicate with other computers to cabinets to put them in. They are regarded as peripheral to the box of complex circuitry (boasting perhaps a panel of switches and lights) that constitutes the basic PDP-11 computer in your system.

Even if your PDP-11 system is very small, operated exclusively from a single terminal (a *standalone* system), it almost certainly includes more than we have so far described. Let us examine the operation of the terminal. It is similar to a typewriter, but with a few special features. There are several different types of terminal; some display characters typed by the user, and the output of the computer, on a TV-style screen instead of on the more common paper roll. There are certain special characters located in different places on different terminals, so you should spend a few minutes in becoming familiar with the characteristics of any new or unfamiliar model.

As on a regular typewriter, there is a SHIFT key. It should be observed, however, that many terminals have only uppercase (capital) letters available. This need not trouble the programmer, since computer instructions do not distinguish between uppercase and lowercase letters. With these terminals, do not use the SHIFT key to obtain regular letters, because other characters will sometimes result. For example, on some terminals, @ is SHIFT-P,] is SHIFT-M, while [is SHIFT-K. Anything of this kind is normally clearly marked on the respective keys.

Be careful always to distinguish: I (capital i), l (lowercase L), and 1 (one); O (capital o) and 0 (zero); and parentheses (.), angle brackets < . >, and square brackets [.].

An important feature is the CONTROL key. Like SHIFT, this does nothing on its own but, when held down while other keys are struck, it produces a whole new set of characters. Some CONTROL characters are just plain characters. If you type CONTROL-A, you will just see ^A appear on the paper or screen. If, however, you type CONTROL-l, no symbol will appear, but generally there will be the same effect as if a tabulator key had been pressed; tabs are normally set at every eighth space. In most PDP-11 systems, if you type CONTROL-C while a program is running, ^C will appear and, in addition, the program will stop (if calculation is in progress, two CONTROL-C are needed for this effect). Several other CONTROL characters have “break” or “interrupt” functions, which may be studied individually as they are needed.

In this book, CONTROL will be denoted by the ^ symbol. *Warning:* This is not the “circumflex” or, on some keyboards, the “up-arrow” symbol (this symbol is often SHIFT-N). Typing ^, followed by C, will also appear as ^C but will not have the special effects of CONTROL-C.

In this book ^A etc. always means type the character while holding down CONTROL, unless specifically stated otherwise.

With the *on-line* switch in the "on" position so that there is communication between terminal and computer, type ^I. As already mentioned, in most systems a tab will be produced at the terminal. Now turn the on-line switch off, disconnecting the terminal from the computer. The terminal still functions as a typewriter, but you may find, depending on what kind of terminal you are using, that now ^I does nothing. Some modern terminals, especially screen terminals, have built into their circuitry the power of responding with a tab to ^I; with others this is not the case, and ^I will not work when the terminal is off-line. A complex process is then necessary to create a tab when the terminal is on-line. The user sees nothing of this process (it is *transparent* to the user). In the on-line state, anything the user types at the terminal (user *input*) goes at once to the computer. A special program, permanently available in the computer, checks this input; the program is set up to respond to ^I by making the terminal type-head or cursor move on the appropriate number of spaces. When the terminal is on-line, there is no direct communication between keyboard and paper or screen. When a printing character is typed, it is the same program, not any mechanical response of the terminal, that causes its *output* to the terminal paper or screen (*echoes* the character).

This program is called the *monitor*. The monitor's function is not confined to printing characters at the terminal. It runs the user's programs, finds space on the disk for records of programs or of data, organizes input and output, and much else. In a timesharing environment, the monitor allocates computer time among the various users. Essentially, it is the control system of the computer.

As an integral part of the computer system, but being a program rather than a physical device, the monitor is an example of *software*. Much of the monitor's job consists of running other programs kept permanently in the computer to perform tasks for the user. These are the *systems programs*, also part of the software; among these we shall, in particular, study editors, assemblers, and loaders. The monitor and various systems programs with which a computer installation is furnished are collectively described as its *operating system*.

It is perfectly possible to use a PDP-11 computer without any operating system at all but, once familiarity has dissipated the sense of power, it is indescribably tedious. Without an operating system, the task of getting a program into the computer and running it is lengthy and involves much the same sequence of steps each time it is done; in other words, it is precisely the kind of task for which a computer is ideally suited. So a computer program to do this job of running programs should be written and, if this "superprogram" is left running permanently, we have a rudimentary operating system. This way the task of getting a program into the computer is only performed once (why even once?); thereafter, the superprogram takes over the job.

As indicated, sufficiently expert persons can write their own operating systems. In this way, a system can be tailored to fit precisely the needs of its users. There are, however, many operating systems designed for use with PDP-11 computers. Many computer purchasers find that their needs are adequately met by one of the commercially available systems, supplied and installed ready to work as soon as the machine is switched on (*turnkey* systems). Otherwise it is possible to purchase separate components and, armed with the necessary expertise, put together a complete system (a *modular* system) from them. Software is generally purchased electronically imprinted on a disk or tape or punched into a reel of paper tape, together with an agreement restricting its use to the purchaser's system only. Software programs are often highly complex, hence expensive to develop, so the inducement to protect them in this way is understandable. It is not at the time of writing wholly clear to what extent software is directly protected by patent or copyright law.

The wide choice of operating systems can create problems when it comes to using an unfamiliar PDP-11 installation. A monitor program will perform a range of tasks in response to suitable character sequences typed in by the user; the monitor has, as it were, a list against which it checks the user's *commands*. The trouble is that different monitors may have different lists, and there is no way to run

a program without knowing the monitor commands in the system. Such differences complicate the whole process of creating programs and may even result in a program that runs perfectly on one system failing to work on another.

Producers of software regularly issue new, improved versions; system managers frequently tinker with their operating systems, sometimes improving them. In the former case, documentation rarely keeps pace with the changes and may be inaccurate when it does; in the latter case, documentation is all too often nonexistent. Consequently, even what purports to be a regular PDP-11 operating system may conceal its mysteries. There is therefore no point in describing every detail of a particular version of a given system here; any reader using a different or amended version would soon give up in despair. Nor, on the other hand, would it be helpful to take refuge in broad statements about the theory of operating systems, because these may be of little use to the student actually trying to run a program.

Our approach is to work at one level of abstraction beyond the details of any particular operating system. Within this chapter, which is devoted to creating a first PDP-11 program and getting it to run, we make use of only a narrow range of operating system facilities. Fortunately, within this range a basic philosophy common at least to most of the various systems available from DEC is discernible. We remain always in close contact with this underlying philosophy, discussing at each stage what is to be expected of the operating system in achieving the task at hand. Our practical illustrations might not work, in the precise form in which we give them, on the system you happen to be using, but you will have the vocabulary necessary to ask informed questions, to seek information in the reference manuals provided with the software and, if required, a sufficient grasp of what is going on to be able to exercise your ingenuity. For example, in Section 1.2 we discuss the editor, the system program that you use to organize text or program instructions into a conveniently accessible form stored on disk or tape (a disk or tape *file*). Every editor has a command that lets you have the next line of your file typed out. With DEC editors this is typically a single letter command. In the editor called EDIT, on which Section 1.2 is based, this command is **L**. The student using, say, the editor TECO, merely needs to check from the manual that the equivalent command there is **T** and make a note of it in the appropriate place in this book.

After working through Chapter 1 you will know enough about your operating system to proceed until you reach input/output programming in Chapter 4. You will, in addition, have little difficulty in running your programs under any DEC operating system you encounter.

Although most PDP-11s are equipped with a DEC operating system or one designed along similar lines, quite a few run under the UNIX system, a product of Bell Laboratories. We do not discuss this system, not only because it is already well documented, but because UNIX, as a system designed to run on many different computers, conceals the individual characteristics of the PDP-11. We are more concerned with enabling the reader to get closer to the machine and make optimal use of those specific PDP-11 characteristics.

In spite of the opening words of this section, we have not forgotten the student sitting in front of a small PDP-11 computer with minimal facilities. Indeed, now that the introduction of the LSI-11 (a microcomputer version of the PDP-11) has made the PDP-11 accessible to hobbyists, there may be many such readers of this book. If this is your situation, you should read Section 4.1 along with Section 1.4 to get your first program running. Section 4.1 has purposely been written with as little dependence on earlier sections as possible, to make it easier for you to get started on the machine.

1.2 THE EDITOR

The function of the editor is, as mentioned in Section 1.1, to help you create and store your files. Some of your files will be lists of instructions to the computer—that is, *programs*. Others may be collections of data to be processed by programs.

Since we do not yet know how to issue instructions to the computer, we cannot write a program; we can nevertheless write a simple file.

Timesharing Systems The monitor will put the editor at your disposal. In timesharing systems, however, the monitor will first check that you have the right to use the facility. Therefore, at the start of each session with the computer, or *job*, you must initiate this checking procedure by issuing to the monitor the command.

LOGIN

On some systems, the appropriate command is **HELLO**. The monitor will not respond to what you have typed until you press the CARRIAGE RETURN key. This is convenient, since you can check that you have made no typing errors before committing yourself. Errors can be rectified by pressing the RUBOUT (sometimes called DELETE) key and then retyping. On a few systems BACKSPACE may be used instead. If you make many errors, it may be easier to delete the entire line you are currently typing and start again. Typing **^U** will have this effect.

The monitor will respond to **LOGIN** by demanding first your identifying number, then your password. Once satisfied with these, it will indicate its readiness to accept your instructions by typing an appropriate message. On some systems the message will be as explicit as **READY**; on others it may be merely a single character, such as . (a period) or **@**. This shows that you are at *monitor command level*. Remember that on most systems you can always return to monitor command level by typing **^C** twice.

On most timesharing systems a job continues, and accumulates connection charges, until you specifically inform the monitor that you wish to terminate it. This must be done by issuing a command, not by switching off the terminal and walking away. A typical command is **BYE**. On some systems you will be asked to **CONFIRM**, to which the response would be **YES**, followed by CARRIAGE RETURN.

EXERCISE: Practice starting and finishing a job until you are familiar with the procedure.

Opening a File At monitor command level, to write a file we must instruct the monitor to run the editor systems program. On some systems there may even be a choice of editor programs. The monitor command to run a systems program is normally **RUN**; on some systems the proper command may be **R**. You must type this, followed by a space, and then the name of the editor program. Thus, to invoke the editor **EDIT**, you would type

RUN EDIT

followed by a CARRIAGE RETURN, which we shall denote by **↵**. On some systems a dollar sign, **\$**, must be prefixed to the name of the systems program when the **RUN** command is used. If so, you would enter

RUN \$EDIT↵

The editor will respond with an indication that it is ready to receive commands telling it with which files it is to deal; this indication may be a single symbol, such as **#**. We want to create a new file, so we must choose a name for it. File names may be up to six characters in length. The first character should be a letter, but the rest may be letters or digits. We may as well call our file **TEST**. File names are rounded off with an *extension*, four characters in length with the first character a period. The purpose of file name extensions is to convey useful information to users as well as to systems programs about the nature of the file. If, for example, our file were an assembler language (macro) program, we

should call it **TEST.MAC**. We are just going to write a file containing a simple text, and no particular extension is specified for this. It is not a good idea, however, to omit the extension, since in some systems a file without an extension is assumed to be written in whatever language the system manager has chosen to encourage. If, for example, the manager prefers the language BASIC, our program **TEST** would be automatically given a *default* extension and would become **TEST.BAS**. To avoid any such confusion, let us make up a suitable mnemonic extension and call our file **TEST.TXT**.

We have to tell the editor that we want to send what we write to the file **TEST.TXT**; that is, **TEST.TXT** is to be *opened* for *output*. So, as our response to the editor's indication (here a **#**) that it is ready to receive such information, we type

TEXT.TXT↵

Now that **EDIT** knows what you want to do with any text you type, it shows that it is ready to receive text editing commands by typing *****.

Not all editors separate in this way the stages of receiving file opening commands as against text editing commands. The editor **TECO**, for example, issues a ***** to indicate readiness for either kind of command. It is not, therefore, adequate just to type the name of a file; you must first tell **TECO** that you are going to open a file for output by issuing the command **EW**.

EWTEST.TXT

Observe that commands are issued to **TECO** by pressing not **CARRIAGE RETURN**, but the **ESCAPE** key (called **ALTMODE** on some terminals) twice in succession. When you press **ESCAPE**, the monitor prints the character **\$** at the terminal. **TECO** is available on some **PDP-11** systems; it is a very elegant and powerful editor, and we recommend it highly.

Writing a File While the editor is in a state of readiness to deal with text editing commands, it reserves a work space for handling any text you wish to manipulate. This work space is called a *buffer*, and it is part of the rapidly accessible information storage area, or *memory*, of the computer. Memory (also known as *core*) is available to the user's own programs as well as to systems programs. Information stored in memory is not retained after a session at the computer is completed; for more permanent storage, disk or tape files must be written.

At the start of an editing session, the buffer is empty. The first thing to do is to *insert* the text we want. Some editors will just treat anything you type at this stage as text for insertion, until you issue a command signifying end of text. Generally, however, a special command such as **I** must be issued. You have to know the exact way in which the command is to be used (the *syntax* of the command). With **TECO**, for example, the syntax of the **I** command is very simple: after typing **I**, you just type all the text you want to insert, followed by **ESCAPE** twice in succession. A session might look like this.

***I**THE QUICK
BROWN FOX
\$\$

The ***** was typed by **TECO**; all the rest was typed by the user. Remember that **\$** appears at the terminal when **ESCAPE** is pressed. Note that the editor treats **CARRIAGE RETURN** as part of the text. After this, the buffer contains

THE QUICK↵BROWN FOX↵

where we have used the symbol $\leftarrow$ to indicate the presence in the buffer of a CARRIAGE RETURN.

With EDIT, the syntax of the I command is somewhat different. To prepare this editor to insert lines of text into the buffer, the I command must first be *entered*.

I $\leftarrow$

EDIT will now treat everything you type (including CARRIAGE RETURN whenever you go on to a new line) as text for insertion until you press the LINE FEED key, denoted here by $\downarrow$. Thus, with EDIT, your session might look like this.

```
*I
THE QUICK
BROWN FOX
```

After the editor's prompting *, you typed

I $\leftarrow$ THE QUICK $\leftarrow$ BROWN FOX $\leftarrow$ $\downarrow$

All of this goes into the buffer, except for the $\downarrow$, which merely indicates the end of the input text. The editor will issue a fresh *, showing that it is ready for more commands.

The buffer now contains all that we planned to write, and we should transfer its contents to the output file. The command EX transfers the contents of the buffer to the output file. It also *closes* the output file and finds a place for it in the user's disk area. With TECO, EX also returns you to monitor command level. With EDIT, EX results in the # prompt, ready for new file opening commands. To remove control from this editor and return it to the monitor, you type ^Z (remember that this is CONTROL-Z); this will work with most DEC systems programs.

Back at monitor command level, type DIR $\leftarrow$ to list the *directory* of your files. As you see, TEST.TXT is there! To find out what is in it, issue to the monitor the command

TYPE TEST.TXT $\leftarrow$

and see for yourself.

EXERCISE: Practice writing files containing various texts. Discover for yourself the effect of RUBOUT, BACKSPACE, and ^U for dealing with errors. What happens if you type ^Z without issuing an EX command first? What is the effect of ^C? What happens if the file name you give for output already belongs to an existing file?

Editing a File Suppose you want to amend what you have written in your file TEST.TXT. First, repeat the procedure discussed previously up to the point where the editor is ready to accept file opening commands. Text manipulation can take place only in the buffer, so we must be able to bring material from our file to the buffer; that is, TEST.TXT must be opened for *input*. The amended material is to go back into TEST.TXT and form the updated version of that file, so TEST.TXT must also be opened for output. With EDIT, this is done by typing, in response to the editor's # prompt,

TEST.TXT<TEST.TXT

and entering the command with $\leftarrow$. Here, TEST.TXT is first named as output file; then, after the < character required by the syntax of this command, also as input file. So if you want the new version

of your file to have a new name, say **TEST1.TXT**, you would instead enter in response to # the command

TEST1.TXT<TEST.TXT

Whichever editor you use, it is important to realize that editing is a three-stage process. Text is brought from input file to the editing buffer, manipulated in the editing buffer, and sent from the editing buffer to the output file. Doing this with the same file for both input and output is such a common procedure that there are often special commands for the purpose. For example, it is possible on some systems with the EDIT editor to enter, at *monitor* command level.

EDIT TEST.TXT

This not only opens **TEST.TXT** for both input and output, but also reads text from that file into the buffer. In addition, when the editing session is completed, the command **EX** will return control directly to the monitor.

Unless some special editing command has been used, however, we are now at the stage of having our file open for both input and output, but the buffer remains empty. All we have done is set up a two-way channel for communication between our file and memory; as yet, no actual communication has taken place. A command must be issued to read text into the buffer. With EDIT, for example, the command is **R**.

After reading our text into the buffer, we can check that it is what we expected by issuing a command to type it at the terminal. With EDIT this command is **L**. If we enter an **L** command, the editor will type for us

THE QUICK

That is, we will be presented with the first *line* of our file. Note that the ↵ recognized by the editor as terminating a line is regarded as part of that line. To get two lines of our file typed, we need the command **2L**, and so forth. Since our file contained only two lines of text, **100L** will have just the same effect as **2L**.

The command **L** types a line of text, starting from the editor's position indicator, or *pointer*. Since text changes can be effected only at the position of the pointer, it is important to be able to move it as one wishes. After reading in text at the start of an editing session, the pointer is positioned by the editor at the beginning of the buffer. Try issuing the command to advance the pointer one line which, with EDIT, is **A**. After this, **L**, or **2L**, will just show us

BROWN FOX

The line immediately before the pointer can be displayed by entering the command **-L**. Note that this does not move the pointer, for which the special command, here **A**, must be used. Any whole number, positive or negative, may be used with these commands.

- EXERCISES:**
- (i) What happens, with the editor you are using, when you enter a command to advance the pointer by more lines than there are lines of text in the buffer?
 - (ii) Replace **TEST.TXT** with a file of the same name containing the text
**THE QUICK BROWN FOX JUMPS
OVER THE LAZY DOG**
 - (iii) Using the **I** command (or its equivalent with your editor) to insert text at the position of the pointer, amend **TEST.TXT** further to read

THE QUICK BROWN FOX JUMPS
AGAIN AND AGAIN
OVER THE LAZY DOG

Can you still use EX with no problems?

- (iv) What do you suppose is the purpose of the “backup” file TEST.BAK you now find in your directory?
- (v) What happens if you type ^C before closing your file with the EX command? Once again, check the contents of your directory.
- (vi) With EDIT, the command K will delete an entire line of text. Any whole number, positive or negative, may be used with K. Amend your file again to read

THE QUICK BROWN FOX JUMPS
OVER THE LAZY DOG
AGAIN AND AGAIN

Check the contents of both TEST.TXT and TEST.BAK.

- (vii) Amend your file once more to read
THE QUICK BROWN FOX JUMPS
AGAIN AND AGAIN OVER THE LAZY DOG

Character-Oriented Commands We now know how to manipulate lines of text. With the commands now at our disposal, we can make any change we wish in a file by deleting lines and inserting the new version. It would, however, be tedious to have to go to such lengths for even the smallest change. Suppose, for example, that we want to insert a period after DOG in the version of TEST.TXT created in Exercise ii. We know how to insert the period, as long as we can move the pointer to the appropriate place. Observe that an editor’s pointer is always regarded as being positioned between two characters, or before the first character in the buffer, or after the last character in the buffer. It is *never* regarded as being positioned *on* a character. We want to insert a period after the G of DOG, but before the ↵ following that word. We must position the pointer at the same place.

Editors normally have a command to search for a specified text and position the pointer immediately after it. With EDIT this command is G. The syntax of the G command is the same as that of the I command. It will seek out the first occurrence of the specified text following the position of the pointer when the command is issued. In the present case, there is no ambiguity in just searching for G. So, with the editor ready to receive text editing commands, we type

G↵G↓

entering the G command with a ↵, specifying G as the text to be sought, and terminating the text with a ↓ (LINE FEED).

We can use the familiar typeout command, here L, to check the position of the pointer. The precise effect of L is to display text in the buffer from the pointer up to and including the next ↵. (What exactly does -L do? What does 0L do?) Note that the line deletion command, K, will delete whatever L preceded by the same number (positive, negative, or zero) would display. Check also the effect of the line advance command, A, when the pointer is not at the beginning of a line.

You should practice using the search command, here G, to move the pointer around. 2G may be used to search for the second occurrence of the specified text, and so forth. (What happens if G is issued, but the specified text is not in the buffer, or if it is in the buffer but before the pointer? Where is the pointer positioned after an unsuccessful search? Where is the pointer positioned after inserting the period following DOG?) There will be a command to reposition the pointer at the beginning of the buffer; EDIT uses B for this.

Suppose we want to change JUMPS to JUMPED. Some editors have a special command in

which one specifies first the text as it is and then the text as it is wanted. With TECO this is even incorporated with a search; with the pointer positioned somewhere before the word **JUMPS** we could just type

FSJUMPS\$JUMPED\$\$

The **FS** command instructs TECO to search for the text specified up to the first **\$** (ESCAPE) character and replace it with the new text that follows, up to the terminating **\$\$**.

Most editors, however, are less helpful. The usual routine is to search for **JUMP**, delete the **S**, and insert **ED**. The **EDIT** command to delete a character following the pointer is **D**, which may be used preceded by a positive or negative number to delete several characters respectively following or preceding the position of the pointer. When using this command, remember that spaces and punctuation marks are also characters. It may be convenient at times to reposition the pointer back or forth a number of characters. **EDIT** accomplishes this with the **J** command, preceded by the appropriate positive or negative number.

- EXERCISES:**
- (i) Create a file containing the text


```
ALL THAT GLISTERS IS NOT GOLD
SHAKESPEARE
1596
```

 and return to monitor command level.
 - (ii) Amend the file to contain


```
ALL IS NOT GOLD THAT GLISTERS
CERVANTES
1615
```

 and return to monitor command level.
 - (iii) Amend the file again to contain


```
ALL IS NOT GOLD THAT GLISTENETH
MIDDLETON
1617
```

 and return to monitor command level.

Suppose we want to amend our original version of **TEST.TXT** by having all the text on one line. We must search for **QUICK**, delete the $\leftarrow$ that now, after the successful search command, follows the position of the pointer, and insert a space. Deleting a $\leftarrow$ presents no problem, as long as we are aware that:

Pressing the CARRIAGE RETURN key produces two characters; first a carriage return, then a line feed.

Carriage return is the mechanism that sends the terminal back to the start of the same line; in the unlikely event that it is needed alone, it may be produced by typing **^M**. A line feed moves the terminal down one line; it is produced by pressing the **LINE FEED** key or, equivalently, typing **^J**.

- EXERCISES:**
- (i) Check for yourself the effects of **^J** and **^M** separately when the terminal type-head or cursor is not at the beginning of a line.
 - (ii) In the original version of **TEST.TXT**, check that there are indeed two characters between **QUICK** and **BROWN**. See what happens if you delete just one of them.

Page Structure The quantity of text read by an **EDIT R** command from the input file to the buffer is called a *page*. This word has here a technical meaning rather than its regular typographic

sense. Our file `TEST.TXT` was so small that we did not need to trouble ourselves about pages. Long files, however, should be divided into pages for two reasons: not all editors take kindly to an attempt to read more into the editing buffer than it will hold; and it is easier to find one's way around in a long file if it is page structured, using page-oriented commands. It can normally be assumed that the editing buffer is large enough to hold the contents of a display screen or a double line-spaced, regular size, typed paper page.

The usual way to indicate to a DEC editor that the end of a page has been reached is to insert a FORM FEED character, equivalently `^L`. This is done with TECO by just typing `^L` whenever a new page is desired, as part of the text string to be inserted with an `I` command. With EDIT, however, one must type a `↓` to finish inserting text, enter the command `F` (so you type `F↓`) to create a FORM FEED in the text, then issue a new `I` command to proceed.

After all editing work on the page currently in the buffer has been completed, it can be sent to the output file. With EDIT, the `N` command sends the contents of the buffer to the output file; it also deletes everything in the buffer and, if there is more text left in an input file, reads the next page of that text into the buffer. Some editors may need more than one command to do all this.

Some editors have search commands that, instead of confining their scope to the contents of the buffer, will search through the whole input file. When using such a command, it is important to know what the editor does with pages of the input file in which the desired text was not to be found. In EDIT the `H` command reads successive pages of the input file into the buffer, searches them, and sends them to the output file if the desired text is not found. The `P` command, however, performs no output but merely deletes the contents of the buffer before reading in the next page. It is convenient to have both commands available, but it is irritating to find oneself using the wrong one by mistake.

Secondary Buffers Many editors maintain one or more buffers besides the editing buffer, which make it particularly easy to move blocks of text around or to insert the same text at different places in a file.

Suppose that a file is being created in which a particular block of text will occur more than once. The first step is to type the text into the editing buffer using an `I` command. Now a special command is used to copy this text into the special buffer. Normally, this command will have much the same syntax as the command to display text at the terminal. Indeed, the two commands are very similar: one writes text into an area of memory, the other writes it at the terminal. With EDIT there is only one such extra buffer, called the *save* buffer. The command `S` to save lines of text in it is used like `L`, except that only positive numbers are allowed with `S` to specify several lines of text. Remember that the text is still in the editing buffer and should be deleted if it is not wanted at that location. Another command, `U` (unsave) with EDIT, is used later to insert the entire contents of the save buffer into the editing buffer, just before the position of the pointer. The contents of the save buffer are unaffected, so the same text may be inserted elsewhere.

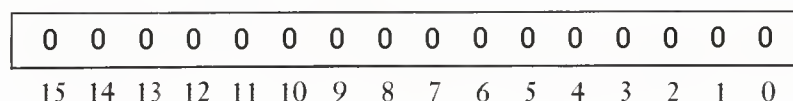
- EXERCISES:**
- (i) Create a file of three pages containing successively the texts `PAGE TWO`, `PAGE THREE`, and `PAGE ONE`.
 - (ii) Without using an insert command, amend your file so that the text on each page corresponds to the order of that page in the file.

1.3 STORING AND RETRIEVING INFORMATION

In Section 1.2 we rather vaguely defined a program as a list of instructions to the computer. Instructions to the computer must be put, by a process that we shall discuss later, into the computer's memory before they can be carried out.

Memory The memory of a computer consists of an extremely large number of small magnetic or electrical devices. For many years the basic device has been a tiny doughnut-shaped ferromagnet (*magnetic core*). Core can easily be magnetized in one direction or the other going around the doughnut, by passing an electric current in the appropriate direction along a wire passing through the hole in the doughnut. More recently, small pieces of semiconductor material (*flip-flops*) that respond electrically rather than magnetically to an electrical stimulus have been used. Whatever the physical form, the basic idea is the same: memory consists of a number of devices, or *bits*, each of which has just two *states* (for a magnetic core, the two states are the two possible directions of magnetization). Electrical connections enable other parts of the computer system to sense the state of any bit (*read* from memory) or to set the state of any bit as required (*write* into memory). It is convenient and concise as well as suggestive of the computational function of a computer to call one of the states 0 and the other 1. It does not matter to us whether 1 means a clockwise or counterclockwise direction of magnetization; such concerns are the business of hardware engineers. Note, however, that while the initial choice of which state to call 0 and which to call 1 is arbitrary, the design of the rest of the hardware must be consistent with it. When we come to do arithmetic, we shall want these 0s and 1s to represent the appropriate numbers. The circuitry that performs addition must be aware that $0 + 0$ is still 0, but that $1 + 1$ is not 1. A bit with 1 in it is said to be *set*, while a bit with 0 in it is *clear*.

There is no great difficulty in encoding information into a collection of two-state bits, as the existence of Morse code makes clear. But not much information can be stored in a single bit. For this reason, computer designers organize memory so that its basic unit, or *word*, is a collection of bits large enough to hold a worthwhile amount of information. Each PDP-11 word contains *sixteen* bits. The state of a word may be indicated diagrammatically.



Note that the bits are numbered 0 to 15 from right to left.

Binary and Octal Notations We have seen how the state of the bits in a PDP-11 computer memory word (the *contents* of the word) may be represented by a sequence of sixteen 0s and 1s. Numerical data, and text such as we wrote in our file of Section 1.2, are examples of the various kinds of information that we need to be able to encode into the contents of a memory word.

Numbers are encoded by letting successive bits represent successive columns in a positional notation. This is, indeed, the normal way in which we represent numbers. We know that 1000 is ten times more than 100 because the 1 is placed one column further to the left in 1000 than in 100. We can write in this *decimal* notation (where ten is the *base* or *radix*) because our system has symbols (*numerals*) to represent all numbers less than ten. Only after we reach 9 in a column do we have to move one column to the left to represent the next higher number.

Since we have only 0s and 1s at our disposal in a computer word, after we reach 1 in a bit we must move a bit to the left to represent the next higher number. That is, we are representing numbers in *binary* notation in which *two* replaces ten as base. In such a system, instead of successive columns, from right to left, denoting units, tens, hundreds, thousands, and so on, they represent units, twos, fours, eights, sixteens, and so on. The word “bit” is, in fact, a contraction of “binary digit.”

For example, since nineteen is equal to sixteen plus two plus one, it is represented by 10 011. Just as with decimal representation, we group digits in threes for ease of reading. We write this succinctly as

$$D\ 19 = B\ 10\ 011$$

where D stands for decimal and B stands for binary. In the binary representation, observe the 1 on the left in the sixteens column, 0 in both the eights column and the fours column, 1 in the twos column and 1 in the units column. Thus, nineteen may be represented in a PDP-11 memory word as

0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1
15		12		9		6		3		0					

Observe that the bit position number gives the power of two represented by a 1 in the bit; we can see immediately that this word contains, in decimal notation, $2^4 + 2^1 + 2^0 = 16 + 2 + 1 = 19$.

The trouble with binary notation is that even quite small numbers are very unwieldy for human beings to read and interpret. For example, not only is it tedious to find the decimal equivalents of 10 010 110 101 and 10 010 101 101, it takes more than a glance even to see that they are distinct numbers. The decimal equivalents are the much more compact 1205 and 1197.

EXERCISE: Check the equivalence of these binary and decimal representations.

Conversion between binary and decimal notations is a tedious business, the job of a machine rather than of a human being. We do, however, need to be readily able to interpret the contents of a word in order to take full advantage of the power of the computer. Fortunately, a good compromise between binary and decimal notations is available: *octal* notation, in which the base is *eight*. Octal representation is easy for human beings to read, and it is very simple to convert between binary representation and octal representation.

First let us see why this conversion is so simple. In octal notation we have the digits 0, 1, 2, 3, 4, 5, 6, 7 at our disposal; after 7, we must move one column to the left to increase a number. Thus, in octal notation, 10 represents the number eight.

$$O\ 10 = D\ 8$$

Since, in decimal notation, $8 = 2^3$, the effect of moving one column to the left in octal notation is the same as that of moving *three* columns to the left in binary notation. Thus, conversion from binary to octal is achieved by replacing successive *triads* of binary digits from right to left by single octal digits. Since a triad of binary digits can represent all numbers between zero and seven (that is, all octal numerals) the replacement is always possible.

B 000 = O 0	B 100 = O 4
B 001 = O 1	B 101 = O 5
B 010 = O 2	B 110 = O 6
B 011 = O 3	B 111 = O 7

For example, B 10 011 = O 23. We worked out earlier than this is D 19, which is also obvious from the octal notation: twice eight plus three equals nineteen.

Consider again D 1205 = B 10 010 110 101. The binary triads are, from right to left, O 5, O 6, O 2, O 2. So the octal representation of this number is O 2265, which means $2 \times (\text{eight} \times \text{eight} \times \text{eight}) + 2 \times (\text{eight} \times \text{eight}) + 6 \times (\text{eight}) + 5$.

To convert octal representation to binary representation, reverse the process. For example, O 734: O 4 = B 100, O 3 = B 011, O 7 = B 111. So O 734 = B 111 011 100. In decimal notation, this is $7 \times (8 \times 8) + 3 \times (8) + 4 = D\ 476$.

It is important to be alert to

$$D\ 10 = O\ 12$$

$$D\ 8 = O\ 10$$

$$D\ 64 = O\ 100$$

- EXERCISES:**
- (i) Why is it so easy to convert between base two and base eight?
 - (ii) Is it equally easy to convert between:
 - (a) Base three and base twelve?
 - (b) Base three and base nine?
 - (c) Base two and base six?
 - (d) Base two and base four?
 In the cases where conversion is easy, describe how it is done.
 - (iii) If you were asked to convert the base seven number 59 to base ten, what comment would you make?
 - (iv) Convert to decimal representation:
 - (a) $O\ 37$; (b) $O\ 40$; (c) $B\ 1\ 111$; (d) $B\ 11\ 110$.
 - (v) Convert to octal representation:
 - (a) $D\ 37$; (b) $D\ 40$; (c) $B\ 1\ 111$; (d) $B\ 11\ 110$.
 - (vi) Convert to binary representation:
 - (a) $D\ 37$; (b) $O\ 37$; (c) $D\ -32$; (d) $O\ -32$.
 - (vii) What is $O\ 100 - 1$:
 - (a) In octal notation?
 - (b) In decimal notation?
 - (viii) Is $O\ 15 - O\ 60$ positive or negative? Why?

The conversion between binary and octal representations takes the computer a negligible amount of time to perform, so systems programs are normally set up to exchange numerical information with the user in octal notation as their normal mode. It is therefore necessary to acquire some facility in using octal notation. This does not mean carrying out complicated octal calculations or conversions between octal and decimal; in due course we shall learn how to instruct the computer to perform such tasks. It is just a matter of getting used to counting in octal. After a little time it will no longer seem surprising that, in octal notation, the result of adding 17 to 762 is 1001, while the result of subtracting 17 from 762 is 743 (check these!). It can occasionally be useful to have a rough sense of how large a number expressed in octal notation is. In our example, since $O\ 1000 = D\ 8 \times 8 \times 8 = D\ 512$, we know that $O\ 762$ is around five hundred; it is not often that we would need to know more.

ASCII Code As well as numbers, we need to encode into computer words letters and, in general, the various symbols that appear on the keyboard of the terminal. There is a comprehensive code in which to every single symbol there corresponds a number. This code, which is widely used on many different machines, is called the American Standard Code for Information Interchange. It is commonly referred to by its acronym ASCII (pronounced az-key). On the full standard terminal there are 127 distinct symbols (this is $D\ 127$). This includes not only uppercase and lowercase letters, numerals and special symbols, but also special combinations such as CONTROL characters, which are regarded as one symbol by ASCII. For example, the ASCII code for $\wedge A$ is 1. Suppose that we have somehow contrived to make a certain word in the computer contain the number 1. By this we mean that, reading from right to left, the first bit in the word is set to 1, and the rest are 0. Depending on what we are doing, we might want this 1 to mean $\wedge A$, or we might want it to mean the number 1. It is important to realize at the outset that the computer cannot "know" what we mean until we instruct it accordingly.

The ASCII code interprets the symbols of the terminal as octal numbers between 0 and O 177. Observe that D 128 is $2 \times (\text{eight} \times \text{eight})$, and so is equal to O 200. Subtracting 1 from this number gives D 127 = O 177 as the number of distinct ASCII symbols.

It is important to understand why $\text{O } 200 - 1 = \text{O } 177$. Consider what happens when we try to add 1 to O 7. The quantity in the units column is increased to eight, so we must carry to the left, into the eights column. So $\text{O } 7 + 1 = \text{O } 10$. Similarly, $\text{O } 17 + 1 = \text{O } 20$. If we try to add 1 to O 77, carrying 1 into the eights column increases the quantity there to eight, so we must carry one more column to the left, into the (eight $\times$ eight) column. So $\text{O } 77 + 1 = \text{O } 100$. Similarly, $\text{O } 177 + 1 = \text{O } 200$, and so on.

Of course, it is possible to convert these ASCII codes into decimal representations, but this is not necessary. It is a much better idea to get used to these numbers in the octal form in which they are normally quoted. Just remember that no digit may exceed 7; and so adding 1 into a column with a 7 in it produces 0, with a 1 carried to the left.

The ASCII codes for A through G are

A	101
B	102
C	103
D	104
E	105
F	106
G	107

What do you suppose is the ASCII code for H? As you have doubtless guessed, it is the code for G increased by 1; which of course means

H	110
---	-----

and so on sequentially through to

W	127
X	130
Y	131
Z	132

The numerals on the terminal have as ASCII codes

0	60
1	61

and so on, through to

7	67
8	70
9	71

It is very convenient that the ASCII codes for successive numerals are themselves successive numbers; observe that one can be obtained from the other by adding or subtracting O 60.

- EXERCISES:**
- (i) How many bits are needed to hold any given ASCII symbol?
 - (ii) Suppose that we want to store, in PDP-11 words, ASCII text that can be in either regular type or italics (sophisticated terminals have more than one typeface available). Suggest an efficient way of storing an ASCII symbol together with the information as to which typeface should be used.
 - (iii) What is the largest number that could be stored in binary notation in the sixteen bits of a PDP-11 memory word?

Addressing Being able to store information in computer memory would not be of much use if we could not retrieve the information when we wanted it. Clearly, there must be some way of distinguishing one memory word from another, and the obvious thing to do is number them. The number by which a word is identified is called its *address*.

We now have two numbers associated with each memory location: the address of the location, and the contents of the location as given by regarding its bit pattern as a number expressed in binary notation. These two numbers are conceptually so different that it may seem unnecessary to warn against confusing them. In practice, however, it is all too easy to do so. Suppose, for example, that we wish to put the ASCII code for **B** into memory location number 2. The correct PDP-11 assembler language instruction to accomplish this is

```
MOV      #102,2
```

Even an experienced programmer, however, is quite capable of thoughtlessly writing

```
MOV      102,2
```

which would set the contents of memory location number 2 equal to the contents of memory location number 0 102. This is a perfectly proper instruction as far as the computer is concerned; it simply does not do what the programmer intended.

The address of a memory location can be expressed in binary notation and encoded as a bit pattern into any memory location. Expressed more succinctly, *the contents of one word may be regarded as the address of another*. We say that the former word *points* to the latter. In assembler language, the symbol @ signifies the concept of pointing. Thus the effect of

```
MOV      @102,2
```

is to replace the contents of the location whose address is 2 by the contents of the location *pointed to* by location number 0 102. This and other forms of address referencing will be discussed more fully (and will seem less mysterious) later on.

PDP-11 memory is organized in a way somewhat different from that of most computers. In the PDP-11, not only does each word have an address, but so also does each half of each word. Each sixteen-bit word is divided into two *bytes* (byte is pronounced like bite) of eight bits each. This allows the convenience of addressing the whole of a word or either half of it, at the cost of some potential for confusion.

The two bytes of any word are given consecutive addresses. The byte composed of bits 0 to 7 (the *low* byte) always has an even number as its address; the *high* byte (bits 8 to 15) of the word has the next higher address. Thus, memory may be viewed diagrammatically as:

Address	Byte		Address
1	high	low	0
3	high	low	2
5	high	low	4
7	high	low	6
11	high	low	10
...	...	...	...
...	...	...	...

Note that *memory addresses are expressed in octal notation.*

The address of a memory word is always the same as that of its low byte. Thus, on the PDP-11,

The address of a word is always an even number.

It is important to remember that the address of the next word is found by adding *two* to the address of the current one. It might seem that problems could be caused by a word and its low byte having the same address, but we shall see that this is not the case.

The Central Processing Unit Suppose that a certain word in memory has its bits set to 0 000 000 001 000 010. Remember that this is merely a statement about the condition of a collection of electrical or magnetic devices. We may wish to ascribe a *meaning* to this bit pattern; a computer, however, will await a specific instruction, rather than intuiting the mental state of the person programming it. So far we have seen that this bit pattern could represent the number 0 102, the memory location whose address is 0 102, or the ASCII code for the letter B. Sometimes the context makes our desired meaning apparent. If the computer is instructed to transmit this bit pattern to the device in the terminal that holds electrical signals for delivery to the printing mechanism (the terminal *printer buffer*), circuitry in the terminal automatically responds by printing the letter B.

The part of the computer that carries out instructions is called the *central processing unit* (CPU). Each kind of operation (such as addition or subtraction) that the CPU can perform corresponds to a specific collection of circuitry. The CPU is induced to select a particular circuit, and thus perform the corresponding operation, by the state of a set of bits in its *instruction register*. If the leftmost six bits are set to 000110, the CPU switches to the circuitry that performs addition; if they are set to 001110, subtraction is selected. We need not be concerned with the electronic details of this process.

The CPU's instruction register contains sixteen bits. The state of these bits specifies to the CPU which operation is to be performed, where it will find the data on which it is to be performed, and where the result should be stored. Observe that the instruction register is the same size as a memory word. This, far from being a coincidence, is a fundamental aspect of computer design.

Instructions to the computer are stored within the memory of the computer.

We can, of course, represent an instruction stored in a memory word as a number in binary or octal notation, just as if it were data. We raised earlier the question of how the computer "knows" what the contents of a memory word represent (is 0 102 data, address, or letter B?). To this we must add a seemingly more fundamental question: how does the computer "know" that a word contains data and not an instruction?

The simple answer is that the computer does not "know." Memory words are just bit patterns to the computer. *Whether the contents of a given word are treated as data or as an instruction is entirely a matter for the programmer.*

The CPU determines which memory word is to be regarded as the current instruction by referring to a register called the *program counter* (PC). A register is a collection of sixteen bits, very much like a memory word, but situated within the CPU rather than in memory and wired to perform a variety of functions.

The CPU does nothing but continuously perform the following cycle of operations.

- 1. Copy the bit state of the memory word pointed to by PC into the instruction register (without affecting the memory word).
- 2. If the instruction now in the instruction register specifies that data has to be fetched from memory:
 - (a) Increment the contents of PC by *two*, to point to the next word.
 - (b) Copy the contents of a memory word (address obtained by reference to the syntax of the instruction and the contents of the word now pointed to by PC) into the appropriate data register.
 - (c) If the instruction's data requirements remain unsatisfied, go back to part a. Otherwise, continue.
- 3. Increment the contents of PC by two.
- 4. Perform the instruction.
- 5. Go to Step 1.

Essentially, the CPU is *programmed* in its hardware to go through the cycle of fetching the instruction pointed to by PC, updating PC, and performing the instruction. Some instructions may require the CPU to get operands from memory (such as an instruction to add the contents of two memory words together); this requires the inner “loop” of Step 2.

It follows that it is up to the programmer to ensure that the CPU does not encounter a word intended as an instruction when it is prepared for data, or vice versa. To a great extent this is taken care of by systems programs. The programmer can, for example, simply write

INC 102

to increment by one the contents of the memory location whose address is O 102. A special systems program, called an *assembler*, will translate this into *two* words of instructions.

increment a memory location
102

The syntax of the instruction to increment a memory location triggers the CPU to perform the inner loop of Step 2 just once. At this stage, PC contains the address of the second word of the illustrated sequence, and the contents of location 102 will be copied into the data register. At Step 4 the contents of the data register will be incremented by 1 and then copied back into location 102.

- EXERCISES:**
- (i) Suppose the instruction INC 102 is stored in memory starting at location 100. What happens when this instruction is performed? What happens if it is part of a loop in the program and gets performed more than once?
 - (ii) The entire instruction INC PC translates into *one* memory word. Will the sequence

INC PC
INC PC

- increase the contents of PC by two?
 (iii) The instruction

```
ADD      #2,PC
```

will add two to the contents of PC. It takes up *two* memory words:

add number to PC
2

Will the sequence

```
ADD      #2,PC
ADD      #2,PC
```

- increase the contents of PC by four?
 (iv) What is the effect of the sequence

```
ADD      #2,PC
INC      PC
```

- (v) The subtraction instruction **SUB** assembles similarly to **ADD**. What is the effect of the instruction

```
SUB      #4,PC
```

in the course of a program?

1.4 RUNNING A PROGRAM

In Section 1.1 we discussed the problems caused by the great variety of operating systems available for PDP-11 computers. These problems will be of considerable concern in this section. It may take a good deal of patience to devise how to run the very simple programs of this section on your own particular PDP-11 system.

No matter how sophisticated a program may be, we can learn nothing about its operation until the results are reproduced in some tangible form. In other words, the program must perform *output* to let us know what has happened. Output may, for example, be in the form of control signals governing the actions of complex machinery or, as we shall consider here, merely printed information at the terminal.

Our first program will instruct the machine to print the letter **B** at the terminal and then stop. On a standalone PDP-11 system the user has direct access to the terminal printer buffer and can simply set its contents to **O 102**; this causes the symbol whose ASCII code is **O 102** (that is, **B**) to be printed out at the terminal. In a timesharing system, on the other hand, it would not be feasible to allow any one user to exercise direct control over the electronic channels through which all user terminals must communicate with the computer. Consequently, in a timesharing system, input and output are entirely under the control of the monitor, except for users with special privileges. In such an environment, the monitor program activates a procedure to check instructions given to the CPU. If your program issues an instruction that tries to move data to your terminal printer buffer, the monitor will intervene, stop

your program, and print a message at your terminal informing you that you have referenced an illegal address.

Operating systems on standalone PDP-11 computers may also have monitor-controlled input and output available, either in addition or as an alternative to leaving it entirely to the user. The monitor routines are efficient and easy to invoke, and programming time is usually saved by using them. We shall concentrate on monitor routines in this section, leaving the discussion of user-controlled input/output (I/O) for Section 4.1.

Monitor-Controlled Output

Our model for I/O routines governed by the monitor will be those available under the RT-11 operating system. This is a broadly applicable choice, since these routines can also be made available on other operating systems. You will need to check whether your system has these routines and, if not, what serves in their place.

The monitor is invoked to send one ASCII character to the terminal by the command

```
.TTYOUT
```

Note that the first character of this command is a period. Since TTY is commonly used as a mnemonic for a computer terminal, the .TTYOUT command is itself reasonably mnemonic.

The monitor must be told what it is to send to the terminal printer buffer. We want to send the ASCII code for **B**. We do *not*, however, write

```
.TTYOUT    102
```

because this would have the rightmost seven bits of memory location **O 102** output as an ASCII character. Instead, the correct form is

```
.TTYOUT    #102
```

We have already encountered the # symbol as indicating that what follows is the actual data (in octal notation) rather than the address where data will be found.

This single command accomplishes all that we planned to do. It is, however, unfortunately not enough to create a file (let us call it **TEST.MAC**) containing just this one line. To understand what more is needed, we must examine the process whereby a program is run.

The Assembler We mentioned in Section 1.3 that the CPU accepts instructions in the form of sixteen bit binary codes. For example, the instruction to add one to the contents of **PC** is

```
0 000 101 010 000 111
```

Even expressed in octal notation as **005207** (check this!), the form of this instruction is by no means convenient for the programmer. As indicated, however, the programmer can instead write

```
INC        PC
```

which is very easy to understand and recall, even if undesirable in its effects (why?). Conversion from this form to the binary code acceptable to the CPU is accomplished by the *assembler*. The assembler is a program that sets up, in memory, a table of *assembler language* instructions, such as the **INC** instruction just cited. Against each entry in the table is the appropriate binary code. In addition, the table must specify how the *operand(s)*, here just **PC**, are to be encoded. Note that in this aspect of its

work the assembler adds nothing to the program; it merely *translates* it from one form to another. Consequently, the entire repertoire of machine instructions is available to the assembler language programmer, and there is never any need to write directly in machine code. There are, however, many occasions when some familiarity with the form of machine code instructions is very helpful.

It is clear that there is nothing inherently unique about an assembler language for a particular machine. Indeed, for some computers several different assembler languages are available; it is by no means clear that this is a benefit. Since the assembler language supplied by DEC as software for the PDP-11 is very rich and imaginative, it is not likely that anyone will be motivated, except as an exercise in programming, to create another. This is especially true since, as we shall see later, this assembler language allows programmers to create new commands to suit their requirements. Such programmer-created commands are called *macros*, and to mark the availability of this facility the assembler language for the PDP-11 is called MACRO-11. Some operating systems, however, offer only a subset of MACRO-11 without macro capabilities; this is called PAL-11, which is acronymic for program assembly language.

Start Address The assembler does not confine itself to translating your program into machine code. It also prepares the ground for the *loader* systems program, whose job is to put the machine code comprising your program into memory, ready to be executed. Among various items of information that the assembler will store in the file it creates to house the binary code translation of your program is the place in your program at which you wish operations to start. This is called the *start address* of the program although, of course, the instruction so designated has no address until the loader sets up the program in memory.

It may seem superfluous to have to specify a start address. It would be reasonable to expect the CPU, when your program is executed, to start at the beginning, go on until it reaches the end, then stop. If this were the case, however, one of the most useful properties of sophisticated loader programs would be lost: the ability to fuse a collection of separately assembled routines into a single program at load time. A loader with this property is called a *linking loader*. This can only be done if just one of the routines marks the beginning of operations with a **START** address.

Labels The way to put a mark against a line of a program is to use a *label*. Programmers must choose their own labels; labels contain up to six letters or digits, of which the first must be a letter. We want to label the instruction `.TTYOUT` as the start address, so **START** is an obvious choice for the label. The label must appear at the beginning of the line it is to mark, followed by a colon, with no intervening space.

```
START:      .TTYOUT      #102
```

There must be some space between **START** and `.TTYOUT`; the assembler is indifferent to the amount of space, both here and at the beginning of lines, so it is convenient to use tabs to align labels and instructions in a pleasing and easily readable format.

The assembler recognizes **START** as a label by the colon that follows it, and enters **START** into its table of labels.

We have not yet specified the line labeled **START** as the start address. All we have done is mark the line in a manner suggestive to ourselves of a start address. The structure of PDP-11 assembler language requires that the assembler be told in the last line of the program what the start address is to be. However, the assembler does not know that the last line has been reached unless it is told so. The last line in *every* program must, therefore, issue the special instruction `.END` to tell the assembler that assembly is now to cease. If `.END` is followed by an expression used as a label in the program, the assembler will mark the line bearing that label as the start address. Thus our program has now become

```

START:      .TTYOUT      #102
            .END          START

```

Note that we format the program so that instructions are vertically aligned.

Types of Instruction Observe carefully that `.END` is not an assembler language instruction. It is an instruction *to the assembler program* that assembly is to cease. Thus, the `.END` line is not itself translated into machine code (it *generates no code*). Statements such as `.END` that, although they appear in a program looking much like instructions destined for the CPU, actually control the assembly process are called *pseudo-ops*.

In fact, `.TTYOUT` is also not an assembler language instruction. It is our first example of a macro but, being a product of the operating system and not of an individual programmer, it is called a *system macro*. The assembler will translate `.TTYOUT` into a whole assembler language routine. We must postpone consideration of the `.TTYOUT` routine, except to remark that it invokes the assistance of the monitor.

The assembler will not, of its own accord, search the file in which system macros are stored. It must be instructed to search for each system macro to be used, by the `.MCALL` pseudo-op. Without this, the assembler will regard `.TTYOUT` as merely an undefined symbol. There would seem to be no advantage gained by this reluctance on the part of the assembler program, and it is to be hoped that future versions will be amended accordingly.

Our program has now grown to

```

START:      .MCALL      .TTYOUT
            .TTYOUT      #102
            .END          START

```

As it stands, this program will assemble perfectly well. Consider, however, what would happen if this program were run. The CPU would begin performing instructions at the start of the software routine representing `.TTYOUT #102`. At the end of this routine the letter **B** would have been printed out, and our program have accomplished all we wanted of it. Unfortunately, there is nothing in the program to tell the CPU that the program execution should be terminated and control returned to the monitor. This is an essential step (why?) and is accomplished by the `.EXIT` system macro. The assembler must be warned to look for `.EXIT` in the system macros file. This can be done by including an additional line.

```

.MCALL      .EXIT

```

Alternatively, this can be done as in the following program. Note that system macro calls and pseudo-ops bear the distinction of a period as their first symbol.

Finally, the complete program to print out the letter **B** is

```

START:      .MCALL      .TTYOUT,.EXIT
            .TTYOUT      #102
            .EXIT
            .END          START

```

You should create a file `TEST.MAC` consisting of this program.

The Assembly Process Your system may have special concise commands to assemble, load and run assembler language programs. You can investigate these at your leisure, but we shall confine ourselves here to a step-by-step approach of more general applicability.

The first step is to run the MACRO systems program. You may wish to refer back to the discussion of running systems programs in Section 1.2. A likely form of the command is

```
RUN $MACRO ↵
```

On one system that we used while developing this text, however, there were two versions of the MACRO assembler, referenced as \$MACRO and \$MACROL; only with the latter was it possible to access the system macros file for .TTYOUT and .EXIT. We mention this merely as a further example of the problems that can arise.

The assembler will indicate its readiness to receive commands by printing a * symbol. We wish to instruct the assembler to take TEST.MAC as input, or *source* file, and create the machine code translation (the *object* file) as output file. It is most convenient for the machine code file to bear the same name with a different extension. We specify to MACRO first the output file and then the input file, separated by an = sign.

```
TEST=TEST ↵
```

If all goes well, the assembler will create the binary file, give it the standard extension .OBJ, and store it in your disk area. You can check the presence of TEST.OBJ in your directory after typing ^Z to return to monitor command level.

Depending on the options available in your system, it may be possible to replace this procedure with the single monitor command

```
COMPILE TEST ↵
```

Compile is more properly used to describe the process of obtaining a binary code file from one written in a higher-level language. On some systems, however, use of the COMPILE command extends to assembler language programs. Note that file name extensions are not given when invoking the assembler program, but the MACRO assembler will only accept TEST as specifying a source file if it finds TEST.MAC in your disk area.

The System Macros Library The assembler may respond to the procedure just discussed with an error message to the effect that there are undefined symbols in your program. These, in spite of our precautions, will be .TTYOUT and .EXIT. Possibly the MACRO assembler you used was a version incompatible with the format of the file in which the system macros are stored (the system macros *library*). To determine this, you will need to consult your system manager. Alternatively, the system macros library may have been unavailable to the assembler. In this case you should ask your system manager to copy it into your disk area; on most systems it is called SYSMAC.SML, but it may in any case be described as the RT-11 system macros library.

Listing File When there are problems with an assembler language program, it may be very useful to study the code produced by the assembler. This, however, is not possible with the object file; getting the monitor to type it out will not help (why not?).

The assembler will, if suitably instructed, produce a readable version of the object file. This is called the *listing* file, and it is obtained by specifying to the assembler that there should be *two* output files.

```
TEST,TEST=TEST ↵
```

The second TEST is interpreted as a demand for a listing file with that name, and the assembler will create TEST.LST and store it in your disk area.

```
.MAIN. RT-11 MACRO VM02-11 26-JAN-79 09:28:07 PAGE 1
```

```
1          .MCALL .TTYOUT,.EXIT
2 000000    START: .TTYOUT #102
3 000010    .EXIT
4      000000'    .END      START
```

```
.MAIN. RT-11 MACRO VM02-11 26-JAN-79 09:28:07 PAGE 1+
SYMBOL TABLE
```

```
START 000000R
. ABS. 000000 000
      000012 001
ERRORS DETECTED: 0
FREE CORE: 18899. WORDS
```

Figure 1.1 A program to perform output.

Figure 1.1 shows a typeout of the listing file for our program **TEST.MAC**. The first column of the typeout gives the line number for each line of your program. This information is provided regardless of whether a given line generates code. The second column gives addresses for the code, which itself appears in the third column. Note that these are in octal notation.

It would appear from this listing that the start address is to be location 0. The address column does not, however, specify the actual address at which an instruction will be loaded but merely the address *relative to the start address*. The latter is, therefore, given relative address 0. It is not the function of the assembler to specify load addresses.

The mystery of what code is generated by system macros such as **.TTYOUT** and **.EXIT** may be resolved by instructing the assembler to list macro expansions with the pseudo-op

```
.LIST      ME
```

although this will not help you much at present.

Observe that the listing file contains no indication of the name of the file. This can be rectified by incorporating the **.TITLE** pseudo-op in the program. **.TITLE** is followed by the name that we want to appear in the listing, which may as well be the name of the file housing the program (but need not be).

```
.TITLE     TEST
```

If there is no **.TITLE** statement, the assembler provides the name **.MAIN**, as you see.

- EXERCISES:**
- (i) Assemble **TEST.MAC**, or your system's closest possible equivalent to it, without errors.
 - (ii) Review from the start of this section, discovering for yourself the result at this stage of leaving out any one or more steps that we have claimed to be essential.
 - (iii) Does it matter at what point in the program the **.MCALL** pseudo-op appears? The **.LIST** statement? The **.TITLE** statement?

The Linker If you are using an RSX-11 or IAS operating system, you should read this section in conjunction with your system's task builder documentation. Users of RT-11 and RSTS operating systems will find this section more closely applicable to their needs.

As we mentioned earlier, the assembler does not determine where your program will be loaded in memory. It merely provides addresses relative to the start address, together with an indication that the number given is a relative address rather than an *absolute* one. Relative code is indicated by the symbol ' in the third column of the listing file. Note in Figure 1.1 that the reference to **START** in the **.END** statement indicates relative code. This is, in fact, the only such code in our program.

The next step in running our program is to create a disk file derived from **TEST.OBJ** in which all relative code is replaced by absolute code. This is the task of the **LINK** program, so we issue a suitable command to the monitor.

```
RUN $LINK↵
```

LINK will respond with a *, and we enter output and input file specifications, just as we did with **MACRO**.

```
TEST,TEST=TEST
```

LINK operates on files with the extension **.OBJ** and so will search out **TEST.OBJ** in your disk area as input file. As the first of the two output files specified, it will create **TEST.SAV**. This is the *memory image* file, a picture of what memory will look like when the program is eventually loaded. In it a start, or *transfer*, address has been set, and all relative code has been *relocated* accordingly. Monitor programs normally require the first 400 (always octal) words of memory for control purposes, so the start address will generally be location 1000. (Why not 400? Why not 1002?)

The second output file created by **LINK** presents its operations in readable format. It is called the *load map* file and is given the extension **.MAP**. You should have the file **TEST.MAP** typed out and examine it.

It remains only to load the memory image file into memory and start execution at its start address. This is a simple task, accomplished by a single monitor command

```
RUN TEST↵
```

whereupon the letter **B** will appear at the terminal. On the RT-11 system we can break the effect of **RUN** into its component parts

```
GET TEST↵
```

which loads the program into memory, and

```
START↵
```

which sets **PC** to contain the start address, thus beginning execution of the program.

- EXERCISES:**
- (i) Run **TEST.MAC** or your system's closest possible equivalent to it.
 - (ii) Write a program to print out the message
TESTING OUTPUT
followed by a ↵.
 - (iii) What is the effect of failing to include **.EXIT** in your program?

2 Fundamentals

2.1 THE REGISTERS

We have described registers as similar to memory locations but physically situated within the CPU and wired to perform special functions. The number of registers varies among the different models of PDP-11 computer, but this variation is concealed by monitors from the general programmer (is *transparent* to the programmer). As far as each user is concerned, there are always eight registers available. In a timesharing system the monitor will continually transfer control from one user program to another. This does not prevent each program from using the same registers, because the first thing the monitor program does when it takes the CPU from a user program is save the register contents in memory, restoring them when it returns control.

Let us modify the program of Section 1.4 to illustrate use of a register. We shall put the ASCII code that we want to output into the first register (register 0) and then have the contents of register 0 output to the terminal.

The assembler recognizes a register by the symbol %. Thus, register 0 is referred to as %0 in a program, and we can have its contents output to the terminal by

```
.TTYOUT    %0
```

It has, however, become common practice to refer to the first six registers, registers 0 to 5, by the names R0 to R5. (Registers 6 and 7 have special functions and are named accordingly.) For ease of communication among PDP-11 users, this convention should be followed. Unfortunately, many versions of the assembler do not recognize these names. It may therefore be necessary to declare to the assembler that R0 is to be the same as %0, and so on. This is done by including in the program the *assignment statement*

```
R0 = %0
```

and similarly for other registers used in the program. There should be no space before =.

```
R0 = %0
...
...
.TTYOUT    R0
```

Before we can output from R0, however, we must move the appropriate data into it. The instruction to move data to a location (register or memory) is MOV. The instruction is followed first by the *source* of the data, which may be an actual number or a location whose contents are the data. Then comes a comma and the *destination* for the data. We wish to put 0 102 into R0.

```
MOV        #102,R0
```

REGTST RT-11 MACRO VM02-11 26-JAN-79 09:34:07 PAGE 1

```

1          .TITLE  REGTST
2          .LIST   ME
3          .MCALL  .TTYOUT,.EXIT
4
5          000000      R0=%0
6
7 000000 012700 START: MOV      #102,R0
8          000102
9 000004 110000 .TTYOUT R0
000004 110000 .IIF NB <R0>,      MOV      R0,%0
000006 104341      EMT      ^O341
000010 103776      BCS      ^-2
9 000012      .EXIT
000012 104350      EMT      ^O350
10      000000'      .END      START

```

REGTST RT-11 MACRO VM02-11 26-JAN-79 09:34:07 PAGE 1+
SYMBOL TABLE

```

R0      =%000000      START      000000R
. ABS.   000000      000
          000014      001
ERRORS DETECTED: 0
FREE CORE: 18895. WORDS

```

Figure 2.1 A program to perform output through a register.

The original contents of **R0** will be destroyed. A listing of the complete program appears in Figure 2.1

- EXERCISES:**
- (i) Use a different register in place of **R0** to find out how the fact that a given register is to be the destination is encoded.
 - (ii) Can any of the eight registers be used in the preceding program?
 - (iii) What is the difference between **MOV #102,%1** and **MOV #102,1**? How does the latter instruction assemble? Will it work?
 - (iv) Without modifying the preceding program in any other way, insert one additional instruction so that **C** gets printed out instead of **B**. (You already know two different ways in which to do this.)
 - (v) There is no reason why a program should not modify its own instructions, although this is more often done for entertainment than utility. Consider the effect of


```

START: INC      LABEL
LABEL:  MOV      #102,R0

```

 Which instruction should follow this to get **B** printed out?

Monitor-Controlled Input We could not consider input in Section 1.4 because we had no way of specifying a location in the machine as destination for a character typed in at the terminal. (Why was this no problem with the source of data for output?) Now, however, we have the registers available to house input characters. The RT-11 system macro to input a single character typed in at the terminal is

.TTYIN

This instruction is followed by the location into which the monitor is to put the ASCII code of the character. For example,

.TTYIN R0

Any previous contents of R0 will be destroyed. The same effect is achieved by simply

.TTYIN

since, if no destination is specified, R0 is assumed. In fact, .TTYIN always results in input to R0. If another location is specified, the input character is then moved by the monitor from R0 into that location. Thus, after

.TTYIN R1

both R0 and R1 contain the input character, since the MOV instruction leaves the source location unaffected. Similarly, .TTYOUT always outputs characters via R0. Thus, after

.TTYOUT R1

both R0 and R1 contain the character to be output.

If no character has been typed when the monitor encounters a .TTYIN call, the monitor will wait. It will wait, in fact, while you type in a whole line of characters and will not continue with program execution until you type a ↵. Only then will the monitor make its *input buffer*, now containing all the characters you have typed, available for program use. The monitor will respond to your pressing the RUBOUT key by deleting the previous character from the input buffer, enabling you to amend any mistakes before entering a line of input. It will respond to ^U by deleting the whole current line from the input buffer.

- EXERCISES:**
- (i) Write, assemble, and run a program to input a character typed at the terminal into R0 and then output the character from R0 to the terminal (*echo* the character).
 - (ii) Amend your program so that it prompts with ? when it is ready to receive input.
 - (iii) Amend your program so that you can type any *two* characters (followed by ↵) and have them echoed by the program at the terminal. Can you echo them in reverse order? What happens if you input just one character, followed by ↵? (*Hint:* Be careful about the use of R0.)
 - (iv) Can you write a program that prompts with ? for you to type in a single character followed by ↵, echoes the character, and then goes through the whole process once more?

Let us write a program to add 1 to a number to be chosen by us at execution time. The heart of the program will be the sequence

```
.TTYIN
INC      R0
.TTYOUT
```

Remember that **R0** is understood with **.TTYIN** and **.TTYOUT**. You should incorporate this sequence into a complete program and check that the program responds to typing in, say, 5 (followed by $\leftarrow$) by printing out 6.

Consider what happens when we type in 5. The monitor call **.TTYIN** results in the contents of register 0 being set to **O 65**, the ASCII code for 5. Then **INC R0** causes 1 to be added to it, yielding **O 66**, so the **.TTYOUT** monitor call causes 6 to appear at the terminal.

Note that this program does not manipulate the numbers we type in; rather, it manipulates their ASCII codes. It will add 1 perfectly well to numbers 0 to 8. But if we type in 9, it will return the character whose ASCII code is **O 72**. Try it and discover for yourself which character has that code. If we type in 10, the first character we type is 1; the program will therefore print out 2. The 0 will remain in the input buffer to puzzle the monitor when the program exits; in many systems the result will be an error message. Even if we could somehow carry out the process of adding 1 successively on each digit of 10, this would not achieve the result of adding 1 to the number **D 10** itself. There is no reason why it should; we have given the machine no indication that 10 is the representation of a number in some “positional” notation. Until we do so, 10 is simply entered as symbol 1 followed by symbol 0.

Since the preceding program manipulates ASCII codes, we can enter any symbol. If we enter **A**, then **B** is printed out. **B** would result in **C**, and so forth. Entering **Z** (ASCII code 132) results in **[** (ASCII code 133).

What follows is an incomplete program fragment that does nothing at the terminal. It merely reads a numeral between 0 and 9, which we type in at the terminal, as the actual number, not as its ASCII code.

```
.TTYIN
SUB      #60,R0
```

The second line subtracts **O 60** from the contents of **R0**. If we have typed in 7 $\leftarrow$, its ASCII code of 67 will have been entered by the **.TTYIN** monitor call. The subtraction command reduces this to 7.

Suppose we type in 8 $\leftarrow$; then **O 70** is entered. The subtraction command reduces this to **O 10** which, again, is **D 8**.

With practice, you will soon find yourself virtually automatically using the **SUB** command to convert the ASCII representation of a digit to the number itself.

Of course, to convert a number, say in **R0**, back to ASCII code before using **.TTYOUT**, the quantity **O 60** must be added back on. This is done by

```
ADD      #60,R0
```

- EXERCISES:**
- (i) Write a program that will accept input of *two* single-digit numbers (followed by $\leftarrow$) and print out their sum, as long as the sum is also a single-digit number. (Be careful with **R0**!)
 - (ii) Amend your program so that you can type $m + n =$ followed by $\leftarrow$, where m and n are single-digit numbers, and have the sum printed out.
 - (iii) Write a program that will correctly print out the sum of two single-digit numbers when the sum has *two* digits. (*Hint*: What is the first digit of the sum?) What does your program make of two plus two?

Numerical Output The **ADD** and **SUB** instructions will work with any numbers. As we have seen, however, problems arise when we try to print out our results in the usual positional notation by which we represent numbers. Suppose that our program contains the sequence

```
.TTYIN      R1
SUB        #60,R1
ADD        R1,R1
```

Note that the third line of this sequence has the effect of doubling the contents of R1. If our input is 9, R1 will end up containing the number eighteen. But `ADD #60,R1` followed by `.TTYOUT R1` will lead to printout as follows: `D 18 = O 22`, `O 22 + 60 = O 102`, and 102 is the ASCII code for B (try it!).

In order to print out the number eighteen in the form 18, we must examine the meaning of this decimal notation. The symbol 1 gives the number of tens in the number eighteen; the symbol 8 tells us how many units are left over. If we *divide* eighteen by ten, the result is 1, with remainder 8.

In its most basic form the CPU for the PDP-11 does not possess a division instruction. An extra electronic chip is then needed to make division, and also multiplication, available. Many, but by no means all, PDP-11 installations do have this option. Division and multiplication are considered to be part of the instruction repertoire for the remainder of this section, which you should study even if your installation does not have these instructions. In Section 2.2 we shall learn how to multiply and divide without the aid of these special hardware instructions.

Recall that we have the number eighteen in R1 and want to divide by ten and save the remainder. The division instruction `DIV` is perfect for our requirements:

```
DIV        #12,R0          ;R0 is correct!
```

Note that we put first the number (in octal notation!) by which we want to divide. The number into which we are going to divide *must be housed in a register*. Furthermore, this must be an *odd-numbered* register, *and the preceding register must be referenced in the DIV instruction*; this register must first have its contents set to zero (for example, by the “clear” instruction `CLR`).

```
CLR        R0
DIV        #12,R0
```

The `DIV` instruction will store the quotient in the preceding register (here R0), and store the remainder in the register used to house the number divided into (here R1). So in this case we shall have the number 1 in R0 and 8 in R1.*

- EXERCISES:**
- (i) Write a program to double input numbers up to and including 9 and print out the result. (It is, for now, acceptable to print out the result of doubling, say, 4 as 08. Why does this happen?)
 - (ii) Amend your program so that it trebles input numbers.

To multiply two numbers together, one of them must be housed in an odd-numbered register. The multiplication instruction `MUL` will store the product in that register.† Thus, we can double the contents of R1 by

```
MUL        #2,R1
```

*This is a simplified description of `DIV`. For a full description see Appendix B.

†This is a simplified description of `MUL`. For a full description, see Appendix B.

We can now write a program to form the product of two single-digit numbers input when the program is run. The sequence to input and perform the calculation is

```
.TTYIN    R1
.TTYIN    R0
SUB       #60,R0
SUB       #60,R1
MUL       R0,R1
```

Note that the first number specified in a **MUL** instruction can be given as the contents of any register or, as we just saw, as an actual number. The same is true of the divisor in a **DIV** instruction.

EXERCISES: (i) Expand the preceding sequence into a complete program.
(ii) Could the order of the first two instructions in the sequence be reversed?

To print out numbers of possibly more than two digits, the process of dividing by ten and saving the remainder must be repeated the appropriate number of times. Suppose we know that a number may consist of at most four digits; in that case, we must divide the number by ten three times over then print out the final quotient followed by the three remainders in the proper order. For example, if our number is **D 2174**, successive division by ten gives:

	Quotient	Remainders		
	2174			
After first division	217	4		
After second division	21	7	4	
After third division	2	1	7	4

Suppose that we start with our number in **R1**. The **DIV** instruction will then use both **R0** and **R1**. We will need three more registers for the remainders; let us use **R2** for the remainder giving the hundreds digit (1 in our example), **R3** for tens, and **R4** for units. Our printout routine therefore begins with

```
CLR       R0
DIV       #12,R0
MOV       R1,R4
```

remembering that **DIV** leaves the remainder in **R1**, where the number was originally housed. We prepare for the next division with

```
MOV       R0,R1
CLR       R0
```

and store the “tens” remainder in **R3**.

```
DIV       #12,R0
MOV       R1,R3
```

Finally, we obtain the “hundreds” remainder in **R2** and print out the entire number.

```
MOV       R0,R1
CLR       R0
```

```

DIV      #12,R0
MOV      R1,R2
ADD      #60,R0
.TTYOUT  R0
ADD      #60,R2
.TTYOUT  R2
ADD      #60,R3
.TTYOUT  R3
ADD      #60,R4
.TTYOUT  R4

```

- EXERCISES:**
- (i) Write a complete program to print out the number D 2174, which should be in R1 at the start of the printout routine. (Use your ingenuity to get it there.)
 - (ii) Next to each instruction of your program write down the contents of the first five registers after the instruction is performed.
 - (iii) Note how we saved a MOV instruction in our printout routine by leaving the final quotient in R0 rather than moving it to R1. Can you make any further economies?
 - (iv) Write a program to accept input of four single-digit numbers and print out their product. Can you make the program work if you type in the numbers separated by spaces? What happens if you type in only three numbers, followed by ↵? What happens if the product contains less than four digits?
 - (v) Using R0 to R5 only, what is the largest number that you can print out?

Numerical Input Numbers of more than one digit are input by a reversal of the process used for their output. Instead of breaking the number down into its separate digits by divisions by ten, we build the number up from its digits by multiplications by ten. Suppose that we want to be able to type 2174↵ and have this read by our program as a decimal number. First, our program reads in the number 2, with a sequence such as

```

.TTYIN   R1
SUB      #60,R1

```

If no more digits followed, we would have finished. This is, however, not the case here; so when our program encounters the 1, it must respond by shifting the 2 one column to the left in decimal notation and adding in the 1.

```

.TTYIN   R0                      ;get next digit
SUB      #60,R0
MUL      #12,R1
ADD      R0,R1

```

At this stage, R1 contains D 21. (What does R0 contain?) Another digit, 7, follows, so we repeat this four-line sequence, after which R1 contains D 217 (why?). One more digit follows, so the sequence must be repeated once more, finally establishing D 2174 as the contents of R1. Our method is to form in sequence the decimal numbers

$$\begin{aligned}
 & \qquad \qquad \qquad 2 \\
 (2 \times 10) + 1 &= 21 \\
 (21 \times 10) + 7 &= 217 \\
 (217 \times 10) + 4 &= 2174
 \end{aligned}$$

Note that although the numbers in successive lines of the table are different, the *process* of going from one line to the next is the same every time: multiply by ten and add the next digit. This is why we could just repeat the same four-line sequence of computer instructions to accomplish it.

- EXERCISES:**
- (i) Write a complete program to accept input of a four-digit number and output it.
 - (ii) Does your program allow you to input a three-digit number?
 - (iii) Write a program to accept input of two two-digit numbers and print out their product.
 - (iv) Write a program to accept input of a single-digit number followed by a three-digit number and print out:
 - (a) their product:
 - (b) the quotient when the latter is divided by the former.

Comments In longer programs it can be helpful to include brief notes explaining the purpose of a line or a collection of lines. This is particularly useful if programs written by one person are to be read by another. To include comment on a line, precede the comment by a semicolon, just as we did on p. 29. A whole line of comment must begin with a semicolon as its first nonspacing character. For example, a program might include

```

;now follows the printout routine
PRINT:      DIV          #12,R0                ;O 12 = D 10

```

How much comment should be included is to some extent a matter of individual taste. Few programmers would include as much as there is in this example. After all, choosing the label **PRINT** obviates further comment on the purpose of the routine. Some programmers would include the comment on division by **O 12**; for others, such a frequently occurring line requires no comment. Certainly the comment in a line such as

```

PRINT:      DIV          #10,R0                ;octal printout

```

is worthwhile; otherwise, on later reading, by the programmer who wrote it or anyone else, the natural assumption would be that a blunder had been made. In general, lines of comment should indicate program flow from one stage to another. Individual instructions deserve a comment if their function is not fairly clear, and most certainly if any subtle trickery is involved. It can also be helpful to explain register usage. Although you should develop your own style regarding comments, be sure it conforms to the general principles we have outlined.

We have purposely been very sparing with comments in the programs and routines in this book. These routines are exercises as well as illustrations. Your first approach to each of them should be careful line-by-line study, writing comments where apt for what is clear to you and reserving queries for any instruction whose purpose you cannot fathom. Then, and not before, copy the program as your own file. Try various inputs and see that the program does what it should. If an instruction is still mysterious, discover the effect of omitting or amending it. Do not be satisfied until you understand the function of every single line. Finally, make and keep a copy of the program that is fully annotated with your own comments. This approach will rapidly develop your own program writing skill.

2.2 BRANCH INSTRUCTIONS

As we mentioned in Section 2.1 and, indeed, as you may have discovered for yourself, some PDP-11 computers have no multiplication and division instructions. Our procedure for outputting decimal

numbers depended on successive divisions by ten; since it would be quite intolerable to be unable to output numerical data in decimal notation, we must find some other way of dividing by ten. Even if your installation has **MUL** and **DIV** instructions available, you should study the following alternative procedure, since it introduces extremely important concepts.

Dividing a positive number by ten is a matter of seeing how many times ten “goes into” the number. If we keep taking ten away from the number until no more tens can be subtracted (without giving a negative result), the number of times we have been able to take ten away is the quotient. For example, from seventy-three we can take ten away seven times; since only three is now left, no further subtractions are to be made. Note that seven is the quotient and three the remainder when seventy-three is divided by ten. All this is obvious, but it is stressed because a clear understanding will help when it comes to programming the procedure.

Suppose that the number to be divided by ten is housed in **R1**. We shall keep subtracting ten from the contents of **R1**. Since the quotient is given by the number of times this subtraction is performed, we must keep a count; we shall keep this count in **R0**, whose contents will therefore be increased by 1 every time we subtract ten from the contents of **R1**. Thus we perform the sequence of instructions

```
INC      R0
SUB      #12,R1
```

over and over again, until further subtraction would yield a negative result. This brings us to the most far-reaching power of the computer: the ability to take alternative courses of action, depending on the result of a previous step. This is usually achieved by an instruction that checks the value of some quantity and, depending on the value, either jumps (or *branches*) to another point in the program or does not.

In the present case, we want to continue subtracting tens from **R1** while using **R0** as a count, as long as the subtraction continues to yield a positive (or zero) result. We achieve this with the sequence

```
DIVTEN:  INC      R0
          SUB      #12,R1
          BPL      DIVTEN
```

The Branch if PLus instruction **BPL** will branch if the result of the previous instruction was “plus,” which is defined to mean positive or zero. The instruction in the program to be performed next if the condition for a branch is satisfied is given in the **BPL** instruction. We want to repeat the two-line sequence of incrementing **R0** and subtracting ten from **R1**, so we label the first line of the sequence and specify a branch to that line.

As soon as the result of subtraction is negative, the condition for a branch is not satisfied, and whichever instruction follows **BPL DIVTEN** in the program is performed. We wanted, however, to stop one subtraction *before* the result becomes negative; consequently, the two-line sequence has been performed just once too often, and this must be remedied. Using the instruction **DEC** to DECrement the contents of a location by 1, the whole routine to achieve division by ten is

```
DIVTEN:  INC      R0
          SUB      #12,R1
          BPL      DIVTEN
          DEC      R0                      ;reverse superfluous
          ADD      #12,R1                  ;repetition of loop
```

The quotient is now in **R0** and the remainder is in **R1**, just as with **DIV #12,R0**. Note that, also as with the **DIV** instruction, we must clear **R0** before performing the division (why?).

Note carefully that the **BPL** instruction checks the value resulting from the instruction *immediately preceding* it. Thus, the order of the two instructions in the **DIVTEN** "loop" could not be reversed.

This is a horribly inefficient way of dividing by ten. We shall encounter better methods later.

- EXERCISES:**
- (i) Using the preceding technique for division, write a program to print out the number **D 2174**, which should be in **R1** at the start of the printout routine.
 - (ii) How many instructions does your program actually carry out? Use the computer to check your answer.

Numerical input without **MUL** is quite easy. The process involves multiplication by ten, which can be done by repeated addition. An economical way to multiply the contents of **R1** by ten follows; the comments denote the contents of **R1** and **R2** after performing the corresponding instruction.

```

ADD      R1,R1      ;2X
MOV      R1,R2      ;2X,2X
ADD      R1,R1      ;4X,2X
ADD      R1,R1      ;8X,2X
ADD      R2,R1      ;10X,2X

```

- EXERCISES:**
- (i) Write a program to accept input of a three-digit number, multiply it by seven, and output it.
 - *(ii) Write a program that accepts input of two numbers of two digits each and prints out:
 - (a) The larger of them.
 - (b) The (positive) difference between them.
 - (c) The smaller, a semicolon, and the greater.
 - (iii) If we preceded our **DIVTEN** routine with


```
MOV      #12,R2
```

 we could eliminate reference to the value of the divisor within the routine itself. Write a program to accept input of a three-digit number followed by a single-digit number and print out the quotient when the former is divided by the latter.

Numerical Input At present we are unable to input a number without knowing in advance how many digits it comprises. Now, as we saw in Section 2.1, the process of inputting a number is based on a routine to get the next digit and add it to ten times the number so far computed. Rather than deciding in advance how many digits there will be, and writing out the routine that number of times, we shall use a branch instruction to repeat the routine as a "loop" until there are no more digits left. The entire process is

```

READ:    .TTYIN
          if not a digit, branch to DONE
          SUB      #60,R0
          MUL      #12,R1
          ADD      R0,R1
          BR       READ
DONE:    ...

```

The **BR**anch instruction **BR** always transfers program control to the specified location, without requiring any conditions to be met. For brevity we have used the **MUL** instruction; you can substitute repeated addition for it if necessary.

All that remains is to recognize when a nondigit is input. We can manipulate using BPL instructions to check whether the ASCII code of the input character lies in the digit range, O 60 to O 71. This is awkward, and such a check is more easily made using instructions that we have yet to learn. For the present, we can simplify our task by agreeing to type a specific *separator character* after the last digit of our number. If we are inputting only one number, we shall just type a ↵ after it in the usual way; two or more numbers will be separated by spaces. The point is that both ↵, comprising the two characters

CARRIAGE RETURN—ASCII code O 15
 LINE FEED—ASCII code O 12

and space, ASCII code O 40, are represented by codes less than O 60. So now we can organize our READ routine to begin with

```
READ:      .TTYIN
           SUB      #60,R0
           if less than zero, branch to DONE
```

You should complete the routine yourself, using the Branch if MInus instruction BMI, which branches exactly when BPL does not branch; BMI is the *complementary function* to BPL. Note that, as before, the condition for the branch is based on the result of the *immediately preceding* instruction.

- EXERCISES:**
- (i) Write a program that will prompt with ? for you to type a number followed by ↵, double the number and print out the result, then repeat the process indefinitely. (The problem is to avoid treating the ↵ as part of the next input.) How do you exit from this program?
 - (ii) Write a program that will accept input of two numbers, separated by a space, and print out:
 - (a) Their sum.
 - (b) Their difference.
 In part b, your program should be able to handle negative numbers. (The NEGate instruction NEG replaces the contents of a location with its negative; use a branch instruction to check whether the result is negative and, if so, to print out its negative preceded by a - sign.)

Exercise ii indicates a way of handling negative output: the corresponding positive number is output, preceded by

```
.TTYOUT      #55                ;minus sign
```

Analogously, we can deal with input of a negative number by actually inputting the corresponding positive number, and storing separately the information that it is to be negated. (What, precisely, is the analogy?) Suppose we are ready to receive input of a number that might be negative. We must check whether the first input character is a -; if it is, we shall record the fact by setting the contents of R5 to 1 (having cleared R5 before starting the routine). Thus, after .TTYIN we must compare the contents of R0 with the number O 55. For this we use the CoMPare instruction CMP.

```
CMP          R0,#55
```

The role of CMP is to precede a branch instruction that takes action as a result of the comparison. CMP has no effect on the contents of the locations it references. In the present case, we need the Branch if EQual instruction BEQ to branch to a routine to deal with the minus sign.

```

READ:      .TTYIN
           CMP      R0,#55
           BEQ      MINUS
           continue here with input routine

```

Elsewhere in the program we would have

```

MINUS:     INC      R5
           BR       READ

```

returning to **READ** to take in the next character.

A slightly different way of reaching the same result is

```

READ:      .TTYIN
           CMP      R0,#55
           BNE      PLUS
           INC      R5
           BR       READ
PLUS:      continue here with input routine

```

where **BNE** is the instruction Branch if Not Equal. Which do you think is the better approach? Note that in the routine using **BNE**, the part handling a minus sign branches back to **READ** to input the first digit. Instead, we could just have another **.TTYIN**. What would be the advantages and disadvantages of this?

When the input routine is finished, a positive number is stored in, say, **R1**. The contents of **R5** determine whether the number is to be left positive (if **R5** is clear) or be negated (if **R5** contains 1). We would normally continue with a routine to set the contents of **R1** to their proper value

```

           TST      R5
           BEQ      POS           ;if R1 positive
           NEG      R1
POS:      ...

```

The **TeST** instruction **TST** checks the contents of the specified location against zero, for use by an immediately following branch instruction. As with **CMP**, there is no effect on the location referenced.

- EXERCISES:**
- (i) Write a program to evaluate expressions of the form $m \pm n$, where m and n are numbers and either $+$ or $-$ may be entered at execution time.
 - (ii) Amend your output routine to suppress leading zeros.
 - (iii) Write a program to convert numbers from octal to decimal notation. Have your program print ! and exit if a numeral 8 or 9 appears in the input.
 - (iv) Find out for yourself what

```

CMP      R1,R2
BPL      ELSWHR

```

does. How does that sequence compare with

```

SUB      R1,R2
BPL      ELSWHR

```

in its effects?

- (v) Write a routine to read numbers separated by spaces or $\leftarrow$ and *ignore* all other characters.

Condition Codes The CPU contains an internal register, the *processor status word*, in which it maintains information necessary for carrying out the current instruction. The processor status word is usually referred to as **PS**; this mnemonic is not, however, recognized by the assembler and, indeed, it is not often necessary to refer directly to **PS** in routine programming tasks.

The low-order four bits, bits 0 to 3, of **PS** are the *condition code bits*. In these the CPU, after it performs an instruction, places information about the result produced by the instruction. Bit 3 of **PS** is the *N bit*, which is set by the CPU whenever an instruction produces a negative result, and is cleared by the CPU whenever an instruction produces a zero or positive result.

The CPU performs a **BPL** instruction by checking the N bit. If the N bit is clear, the CPU transfers program control to the location specified in the **BPL** instruction. If the N bit is set, the CPU does nothing at all in response to **BPL** (it *treats BPL as a no-op*), merely proceeding to the next instruction according to its normal execution cycle. You can readily deduce the effect of a **BMI** instruction. Instructions involving an arithmetical operation all affect the N bit. These include **ADD**, **SUB**, **MUL**, and **DIV** (N bit depends on sign of quotient), as well as **INC**, **DEC**, and **NEG**. **CLR** will always clear the N bit.

The **MOV** instruction sets the N bit according to the value of the data to be moved. **TST** sets the N bit according to the value of the contents of the specified location.

The instruction

CMP *X,Y*

where *X* and *Y* are locations, or expressions of the form **#** followed by a number, sets the N bit according to the result when the data given by *Y* are subtracted from the data given by *X*. Thus

CMP **#60,R1**

will set the N bit if the contents of **R1** are *greater* than **O 60** and will clear it otherwise. Note again that the contents of **R1** are not affected. On the other hand,

SUB **#60,R1**

will subtract **O 60** from the contents of **R1** and set the N bit according to the new contents of **R1**; that is, the N bit will be set if the original contents of **R1** were *less* than **O 60** and will be cleared otherwise. This opposition in the effects of **CMP** and **SUB** is clearly a fertile source of error.

Bit 2 of **PS** is called the *Z bit*. It is set by the CPU whenever an instruction produces a zero result, and is cleared by the CPU whenever an instruction produces a nonzero result. The **BEQ** and **BNE** instructions check the Z bit and respond accordingly.

Note that *the branch instructions themselves merely check the state of the condition code bits without altering them*. Thus we can use a **BPL** followed by a **BNE** instruction to check whether the result of an operation was *strictly* positive.

Since branch instructions are the basic tool for structuring programs, it is important to learn with each instruction the effect it has on the condition code bits. You should be sure, before proceeding further, that you know the effect of every instruction you have learned on the two condition code bits so far encountered.

- EXERCISES:**
- (i) Go through several of your programs, marking against each instruction the state of the N and Z bits after it is performed.
 - (ii) The contents of **R1**, in a given program, are known to take only the (octal)

```

        .TITLE  MEAN
        .MCALL  .TTYIN,.TTYOUT,.EXIT

R0=%0      ;current digit
R1=%1      ;current number
R2=%2      ;running total
R3=%3      ;number count
R4=%4      ;holds mean

START:  CLR     R1
        CLR     R2
        CLR     R3
        .TTYOUT #77

READ:   .TTYIN
        CMP     R0,#60
        BMI     SEP      ;if (R0)<60
        CMP     #71,R0
        BMI     SEP      ;if (R0)>71
RE1:    SUB     #60,R0
        ADD     R1,R1      ;mult (R1) by D 10
        MOV     R1,R4
        ADD     R1,R1
        ADD     R1,R1
        ADD     R4,R1
        ADD     R0,R1      ;add latest digit
        BR      READ

SEP:    INC     R3
        ADD     R1,R2
        CLR     R1
S1:     CMP     R0,#15
        BEQ     MEAN
        .TTYIN      ;check for extra separators
        CMP     R0,#60
        BMI     S1
        CMP     #71,R0
        BMI     S1
        BR      RE1

MEAN:   CLR     R1
        CLR     R4      ;div routine (R2)/(R3)
DIVID:  INC     R4      ;will hold quotient
        SUB     R3,R2
        BPL     DIVID
        DEC     R4      ;reverse superfluous
        ADD     R3,R2    ;repetition of loop
        ADD     R2,R2    ;rounding up routine
        SUB     R3,R2
        BMI     PRINT
        INC     R4

PRINT:  write your own printout routine
        then finish the program

```

Figure 2.2 A program to compute the mean of a collection of numbers.

values 3, 5, and 7. Write a routine to cause branches to VALA, VALB, or VALC under these respective circumstances, without changing the contents of R1.

Counting Data Items In all our previous programming examples, the number of items of data was known in advance. As an example of how to escape this restriction, we shall construct a program to read a collection of numbers and compute their mean (average). This program, for a machine without MUL and DIV instructions, is shown in Figure 2.2.

To calculate the mean, we will need to know how many data items were input. This is done by using a register as counter, incrementing its contents by 1 every time a number is read.

We let any nonnumeric character except $\leftarrow$ serve as a separator between numbers; $\leftarrow$ signals the end of input. The program will allow more than one separator character between numbers, or a separator character before the terminating $\leftarrow$. This obviates excessive care at execution time. So, on finding a separator character, the program must go to a routine that discards any further separator

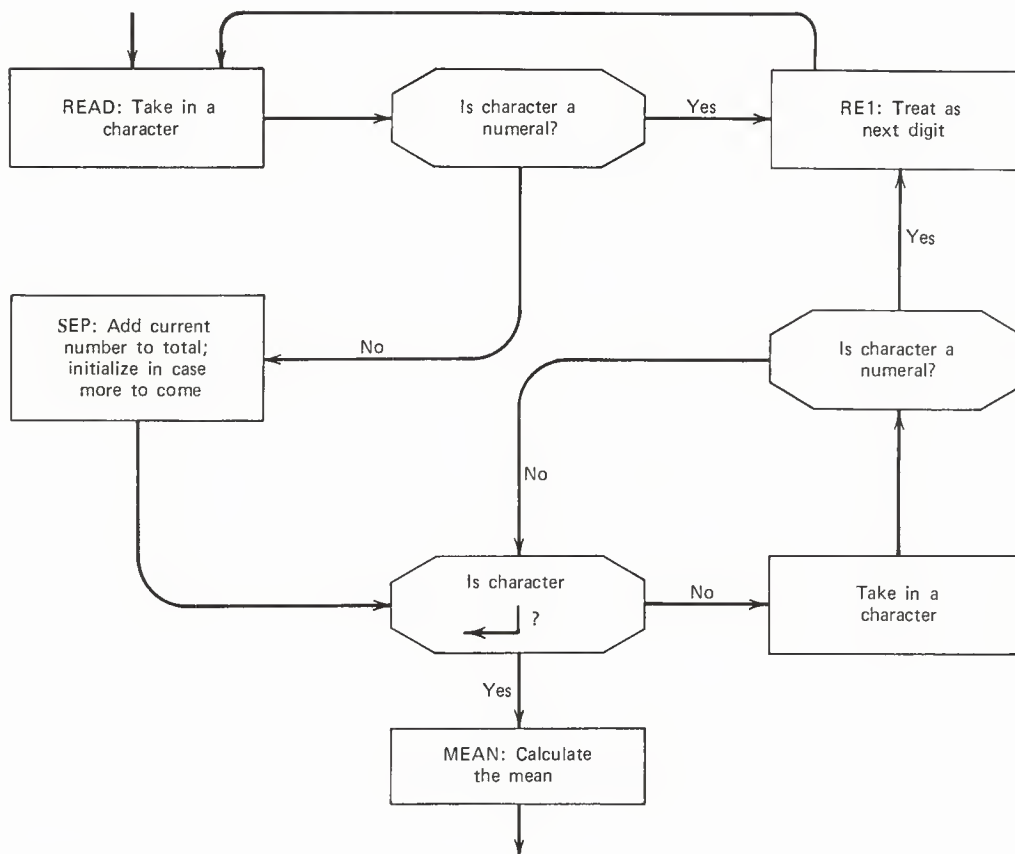


Figure 2.3 A flowchart to accompany the program in Figure 2.2

characters, before attempting to read the next data item. To begin, we set our counter **R3** to contain 0. Our **READ** routine reads a number, using registers **R0** and **R1**. On finding a separator character, routine **SEP** increments the contents of **R3** by 1; adds the contents of **R1** to the running total held in **R2**; resets the contents of **R1** to zero in preparation for reading the next data item; discards any further separator characters; and, if a numeral turns up, returns it to the appropriate point in **READ** (why not return to the start of **READ**?). If **↵** turns up, the program branches to **MEAN** and performs the calculation. The process is illustrated by a flowchart in Figure 2.3.

The mean is computed to the nearest whole number; analyze for yourself how this is achieved. The branch instructions of **SEP** should be studied with especial care.

- EXERCISES:**
- (i) Adapt your **PRINT** routine for the program of Figure 2.2 so that, after printing out the mean of the numbers input, the program again prompts with a **?** for new input.
 - (ii) What happens in the program of Figure 2.2 if you begin input with a separator character? Remedy it.
 - (iii) Write a program to read a collection of numbers and print out the least and the greatest of them.

2.3 MEMORY

In Section 2.2 we considered the problem of reading in a large number of items of data; this was in the program that calculated the mean of a collection of numbers. We could do this with only a few registers at our disposal simply because we did not need to store all the data items separately; we

totaled as we went along. Not all processes, however, can be dealt with in such a way, and much of the power of the computer depends on its ability to store and retrieve large quantities of data. To this end, *memory* (or *core*) is available. Memory is accessible to the programmer in the form of a considerable number of individually addressable computer words. On a timesharing system memory space is needed by many users at a time, so the amount available to each user, although sizable, is not unlimited and should not be unduly wasted.

The memory space needed by your program must be specifically claimed by it. Suppose we want to retain the contents of register **R1** for later use in our program. We must invent our own name (according to the same rules as for choosing labels) for the memory word we want to use: let us call it **MEM**. The instruction

```
MOV      R1, MEM
```

will deposit the contents of **R1** into **MEM**; the contents of **R1** are not affected. The condition code bits are set according to the value of the new contents of **MEM**, just as if **MEM** were a register. In fact, memory locations can be used instead of registers in any of the instructions we have considered. Thus we can move data from memory location **MEM** to memory location **WRD** by means of

```
MOV      MEM, WRD
```

It is not enough, however, just to go ahead and use names such as **MEM** and **WRD** in program instructions. The assembler would print an error message that it had encountered undefined symbols, and it would be impossible to run the program. Instead, we must declare to the assembler that a memory word is to be reserved, and to be given the name **MEM**. We can do this with the line

```
MEM:      .WORD      0
```

The **.WORD** directive tells the assembler to create in the code generated for your program a word with contents set equal to the expression following **.WORD**. Thus the sequence

```
MEM:      CLR        R0
           .WORD      0
           INC        R1
```

would assemble as three words: the code for **CLR R0**, a word of all zeros, then the code for **INC R1**.

The idea, then, is to set aside a word in the program and label it **MEM**. When the program is loaded ready for execution, this word will be assigned a particular numbered memory location whose contents will be set equal to the expression given in the **.WORD** directive. All this is done by the loader program, which will also set **MEM** in its *symbol table* as, by definition, equal to the number of this memory location.

Recalling our discussion of the CPU in Section 1.3, it is clear that we cannot put **.WORD** directives wherever we please in a program. Suppose, for example, we declared **MEM**, as in the preceding example, between two program instructions. After performing the instruction **CLR R0**, the CPU would endeavor to *execute* the word of all zeros created by the **.WORD 0** directive *as an instruction*. A null word is the code for the instruction **HALT**, which stops the CPU in a standalone system and is illegal for the general user in a timesharing system. In any case, we have here the problem, also discussed in Section 1.3, of ensuring that the use of memory to store both instructions and data does not result in their being confused.

We have to position our statements directing reservation of memory space within our program in such a way that the CPU cannot, under any circumstances, encounter them as instructions. All

such statements must precede the `.END` statement, because the latter is a directive to the assembler to stop assembling.

A good place to put `.WORD` directives is between the `.EXIT` monitor call and the `.END` statement. Since `.EXIT` puts the CPU into a routine controlled by the operating system to stop execution of the program, there is no danger that the CPU will proceed to the futile task of executing `.WORD` directives as instructions.

Not every program, however, need have an `.EXIT` call. Indeed, without an operating system there is no `.EXIT` call, in which case `HALT` must be used if the program is to stop itself. There are also programs that run indefinitely, in which case the last instruction in the program will be a branch to elsewhere in the program. Such a program would never reach an `.EXIT`, so there is no point in including one. Here also we can place our `.WORD` directives after the last instruction, immediately preceding the `.END` statement.

It may also be convenient to list the memory requirements of an individual routine immediately after that routine. In Figure 2.2, for example, if routine `SEP` needed memory space, the directives reserving it could be placed immediately before the first line of `MEAN`. (Why does this not cause the problems previously discussed?)

Indexing In Section 2.2 we used conditional branch instructions to deal with the problem of input of numbers comprising varying numbers of digits. The corresponding output problem is somewhat more difficult. We have to divide by ten and store the successive remainders. When division by ten has reduced the original number to zero, we print out the remainders, starting with the last and ending with the first. You should confirm this method by trying it on a few examples. The trouble is that since we do not know how large the number to be output may be, we cannot anticipate the number of locations needed for the successive remainders.

We can overcome this by *indexing*. We can set aside any register—let us call it `Rn`—for use as an *index register*. Register `Rn` will never hold a remainder in the output calculation; rather, it will hold the address of the memory location into which a particular remainder is to go. For example, suppose that in `R1` we have a number that we want to put into memory location `MEM`. We first make sure that the contents of `Rn` are set equal to the address of location `MEM`. Now the assembler will encode any reference to `MEM` as the number that is the address of the location reserved as `MEM`. So we can move this address into `Rn` by

```
MOV      #MEM,Rn
```

Distinguish carefully between

```
(a)      MOV      MEM,Rn
(b)      MOV      #MEM,Rn
```

(a) moves the *contents* of `MEM` into `Rn`
 (b) moves the *address* of `MEM` into `Rn`

After `MOV #MEM,Rn` the instruction

```
MOV      R1,(Rn)
```

puts the contents of `R1` into location `MEM`. The notation `(Rn)` causes the contents of `R1` to be moved, not to `Rn` itself, but to the location whose address is given by the contents of register `Rn`. Distinguish carefully between

- (a) MOV R1,Rn
 (b) MOV R1,(Rn)

(a) moves the contents of R1 into register Rn

(b) moves the contents of R1 into *the memory location pointed to by Rn*

In each case, the contents of R1 are unchanged.

Of course, MOV R1,MEM would be a simpler way to move our data. The power of indexing, however, lies in our ability to increase the contents of Rn at each successive step, stringing out the successive remainders in sequence.

Suppose that the number to be printed out is in register R1. We shall need R0 for the division process, assuming for simplicity that DIV is available. Let us use R2 as index register and locations starting at MEM for the remainders.

We start by setting the contents of R2 equal to the address of MEM. Since R2 is to be used as a *pointer*, we can think of it as starting off pointing to MEM. Each time we divide by ten and deposit a remainder in memory, we increase the contents of R2 so that it points to the next memory word.

Recall that, as we discussed in Section 1.3, in the PDP-11 computer not only each memory word, but also each byte, or half-word, is given a numerical address. Since words have even number addresses:

Making an index register point to the next memory word requires the addition of 2 to its contents.

The process of depositing remainders and increasing the contents of R2 by 2 is repeated until all remainders are found (when is that?). We must now reverse the process, decrementing R2 by 2 and printing out the number contained in the location pointed to by R2 until, finally, the last remainder is printed out. Note how carefully we must ensure that register R2 points to exactly the right place; it is all too easy to be inaccurate in this by one place. Such an output routine now follows. Observe how the L1 loop is ordered to save needing a CMP instruction for the branch. (Could we also do this with the L2 loop?) Note, in the comments, the notation (R1) for the contents of R1; what motivates this convention?

```

L1:      MOV      #MEM,R2
          CLR      R0                      ;quotient in R0
          DIV      #12,R0                  ;remainder in R1
          ADD      #60,R1                  ;convert to ASCII
          MOV      R1,(R2)                 ;and store
          ADD      #2,R2                   ;increment pointer
          MOV      R0,R1                   ;for next division
          BNE      L1                      ;codes set by (R1)
L2:      SUB      #2,R2                    ;decrement pointer
          MOV      (R2),R0                 ;equivalently could
          .TTYOUT                               ;use .TTYOUT (R2)
          CMP      #MEM,R2
          BNE      L2                      ;if more left
          continue here when output finished

```

A program incorporating this routine cannot just direct

```
MEM:      .WORD      0
```

since several words may be needed to house the remainders. The .WORD directive can, however, be

used to reserve several words of storage if the required contents, separated by commas, are specified in the directive. Here five words will suffice (why?), so we can direct

```
MEM:      .WORD      0,0,0,0,0
```

This reserves a block of five words, of which the first is labeled MEM.

Since we are not concerned with the initial contents of the block at MEM, we can alternatively use the directive .BLKW to reserve a BLock of Words. This directive is followed by an *octal* statement of the number of words required.

```
MEM:      .BLKW      5
```

Normally, these words will have their contents initially zero, but this should not be relied on.

- EXERCISES:**
- (i) Write a routine to move the data contained in the location pointed to by R0 to the location pointed to by the location pointed to by R1.
 - (ii) Same as Exercise i, except that R0 is to be replaced by MEM and R1 by WRD.
 - (iii) What would you expect MOV R1,(R1) to do? Does it?
 - (iv) Write a program to read a collection of numbers and calculate and print out how many numbers in the collection are less than the mean of the collection.
 - (v) Write a program that will read a character string comprising uppercase letters of the alphabet; then print out a "coded" version of the string by replacing each letter in the following table with the one below it.

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
T H E Q U I C K B R O W N F X J M P S V L A Z Y D G
```

Modify your program to accept spaces and punctuation marks and print them out unchanged.

Debugging Some of the exercises in this section call on you to write programs of considerable complexity. There will be an input routine containing a loop to move data from registers to memory; exit from the loop must be precisely judged to avoid generating sham data or losing genuine data. An output routine must deal with analogous problems in reverse. And any calculation performed by the program must pick up data from *exactly* where the input routine left them and deposit data *exactly* where the output routine can pick them up. It is often very tricky to code internally complex routines so that they properly coordinate, and, one must confess, it is often rather boring. An unavoidable consequence is that writing a program without errors is a rare and remarkably lucky occurrence. Sometimes the errors are spectacularly obvious. While developing this book we wrote a program in which a routine cleared a block of memory locations, starting at the highest address in the block and counting downwards. R0 held the lowest address in the block, and R2 was used as pointer. We were to exit from the routine by

```
CMP      R2,R0
BPL      REPEAT      ;get more
```

We inexplicably wrote (R0) instead of R0. The contents of the lowest address in the block were less than the address at which the first line of the program had been loaded. So the program was all set to wipe itself out, line by line. Very rapidly, however, the process of destruction reached the offending loop itself and, since we were working on a standalone machine, the CPU stopped (why?).

Errors do not always announce themselves so plainly. It is quite easy, while avoiding a glaring mistake such as the one just described, to set up the loop so that it is performed just once too often, and one location too many is cleared. If this were the last location in a large block reserved for input data, the error would be revealed only by input of the maximum permissible quantity of data; in a program performing complex calculations it might remain concealed even then. Software programs are among the most complex in use, and the more cynical computer professionals claim that error-free software programs do not exist.

Stoic acceptance of the inevitability of errors, however, is small consolation when you are faced with a program that does not run as it should. Most PDP-11 operating systems include the system program ODT-11 (On-line Debugging Technique) for searching out "bugs," or errors. ODT is very useful and is described in Appendix A, which you should begin to study along with Section 2.4.

Some systems have their own version of DDT, the debugging program designed for the DECsystem-10 computer. If this excellent aid is available on your system, you should learn how to use it forthwith. A full description of DDT will be found in Michael Singer, *Introduction to DECsystem-10 Assembler Language Programming* (Wiley, 1978).

Software debugging aids work by letting you execute your program a bit at a time, checking as you go along that it is doing what it should. A primitive version of this *tracing* process is available without the debugging aids. Suppose, for example, you want to check that input is reaching its intended destination in memory. Amend your program so that it runs to the appropriate point and no further by inserting an `.EXIT` at that point. Arm yourself with a *listing* of your program, then run it. The `.EXIT` call will not affect memory contents, and these can now be Examined using the special `E` monitor command. If your program was loaded so that, say, `MEM` is location `O 1376` then, with the RT-11 operating system, the monitor command

```
E 1376←
```

will get the (octal) contents of `MEM` printed out. This way you can check the working of any part of your program. You can have the contents of a block of locations printed out by specifying the first and last addresses in the block, separated by a `-` sign.

```
E 1370-1376←
```

Your program listing will lead you to the load address for `MEM` by telling you the address of `MEM` relative to the start address. Thus, for example, `MEM` may be listed as `000376'` in an instruction referencing it; recall that this is relative to the start address `000000'`; that is, it is *relocatable* 376. The load address is found by adding the address at which the first line of the program will be loaded; this is the *relocation constant*. The relocation constant is normally `O 1000` but, in any case, you are by now well able to write a program that prints out its own start address.

It may sometimes be helpful to test a program by starting it with certain memory locations that have specified contents. The `D` monitor command can be used to Deposit data into memory. The location and its desired contents should be typed, separated by an `=` sign. For example, to set location `O 2000` to contain `O 10`,

```
D 2000=10←
```

Data may be deposited into a block of memory locations by typing the first address in the block, then `=`, and then the desired word contents, separated by commas. For example,

```
D 2002=1,2,3,4←
```

will set location **O 2002** to contain **1**, **2004** to contain **2**, **2006** to contain **3**, and **2010** to contain **4**. Note that both **D** and **E** must specify word (that is, even-numbered) addresses.

Once you are satisfied that a routine is fully debugged, include in your comments a record of exactly what it does. With an input routine, for example, you might note

```
;routine to read list of numbers from terminal, any
;separators, carriage return ends input. Uses R0 for
;input, counts numbers in R1, deposits in memory indexed
;by R2. Requires R1 clear, R2 pointing to first location.
```

A routine annotated in this way can easily be modified for use in other programs.

Automatic Pointer Stepping In the preceding printout routine we used the sequence

```
MOV      R1,(R2)
ADD      #2,R2
```

in a loop routine to “step” through a sequence of memory locations into which data were to be deposited. Data are so very often stored in blocks of consecutive locations that such a stepping process is a frequent programming requirement. PDP-11 hardware is designed to allow pointer incrementing to take place as part of any instruction using a register as index register. Specifically, suppose that **Rn** is being used in an instruction as an index register: that is, **Rn** is to be found in the form **(Rn)** within the instruction. Then, if we replace **(Rn)** by **(Rn) +**, the instruction will be carried out as before and, in addition, the contents of **Rn** will be incremented to point to the next location. Thus, the preceding two-line stepping sequence can be replaced by the single instruction

```
MOV      R1,(R2) +
```

Note the order: first the move is performed, then the contents of **R2** are incremented. The CPU realizes from the form of the instruction (we shall examine this more closely later) that **R2** is being used as a pointer to words rather than bytes; consequently, the increment is **2** rather than **1**, without the programmer needing to take any special action. This way of referring to the location pointed to by **R2** is called *autoincrement addressing*.

Reasonably enough, we also have *autodecrement addressing* to step through a block of memory locations in the opposite direction. This time the notation is **-(Rn)**, indicating clearly that the order is: first decrement **Rn**, then perform the instruction. Thus the new value of the contents of **Rn** will be used in performing the instruction. So, in our printout routine, the sequence

```
SUB      #2,R2
MOV      (R2),R0
```

can be replaced by the single instruction

```
MOV      -(R2),R0
```

As with autoincrement addressing, the amount of decrement is taken care of by the CPU.

We postpone until Section 2.4 discussion of instructions using autoincrement or autodecrement addressing in which two locations are addressed, both via the same register. We suggest that you avoid such forms as **MOV (R2) +, (R2)** until then.

- EXERCISES:**
- (i) Write a routine to go through a block of memory words, setting the contents of each word equal to its address.
 - (ii) Write a routine to move the contents of a hundred-word block starting at **MEM** to a similar block starting at **WRD**. What is the address of the last word in each block?
 - (iii) In a hundred-word block starting at **MEM** the addresses of a hundred data items are stored. Write routines:
 - (a) To add all the data items together.
 - (b) To store each item in the word of the block at **MEM** that previously pointed to it.
 - (iv) One hundred data items, all different, are stored in a block starting at **MEM**. The same data items, in a different order, are also stored in a similar block starting at **WRD**. Write a routine to replace each item in the block at **MEM** with the address in the block at **WRD** where it is also stored.

Sorting To illustrate the use of blocks of memory locations, we shall write a program to read a list of whole numbers and then print them out in increasing numerical order. This calls for a rearrangement of the data items into numerical order. Such programming serves as a good introduction to the frequently occurring problem of arranging a list of words in alphabetical order. There are several different methods available for this *sorting* of data; which is most efficient will depend on the circumstances. The method we shall use is reasonably efficient and fairly straightforward to program.

To arrange our list of numbers in increasing numerical order, think of them as one long list across a page. Starting from the left, we move across the list to the right. On reaching each number, we compare it with its successor to the right. If the successor is greater or equal, we move on one step to the right. Otherwise, we interchange the two numbers, and move one step to the left to see if further exchanges of the smaller number are needed (if we are already at the leftmost number, we carry on to the right). We proceed until we reach the rightmost number.

For example, starting with

2 1 7 5 3

the successive steps are

1	#	2		7		5		3
1		2	*	7		5		3
1		2		5	#	7		3
1		2	*	5		7		3
1		2		5	*	7		3
1		2		5		3	#	7
1		2		3	#	5		7
1		2	*	3		5		7
1		2		3	*	5		7
1		2		3		5	*	7

On each line we have placed, between the two numbers that have just been compared, a # if they have been interchanged and a * if they have not. As you see, when we can compare the last two numbers and find it unnecessary to interchange them, we are finished.

This may seem rather complicated, but the heart of the program is a simple routine **COMPAR** to compare two numbers and possibly interchange them. Suppose we have our data numbers contained in a block of locations starting at **MEM**. We will use register **R2** to keep track of where we are in the

```

MOV      R0, R2
COMPAR:  CMP      R1, R2
        BEQ      DONE
        MOV      (R2)+, R3
        CMP      (R2), R3
        BPL      COMPAR
SWAP:    MOV      (R2), R4
        MOV      R3, (R2)
        MOV      R4, -(R2)
        SUB      #2, R2
        CMP      R2, R0
        BPL      COMPAR
        ADD      #4, R2
        BR       COMPAR

```

Figure 2.4 A routine to sort a collection of numbers into increasing numerical order.

block; that is, as index register. We shall therefore at each stage be comparing the contents of (R2) with the contents of the next word after (R2); remember that (R2), denoting the contents of R2, is the address of our data here.

The entire sorting routine appears in Figure 2.4. For convenience we have assumed in it that the address of MEM has been stored in R0 and the address of the last item of the list in R1.

Sorting problems occur so frequently that you should spend some time thinking out exactly how this routine works. You might like to practice using it to arrange randomly dealt playing cards in numerical order, or Scrabble letters in alphabetical order. Be careful to keep track of the contents of "the registers" as you go. The flowchart in Figure 2.5 may be helpful here.

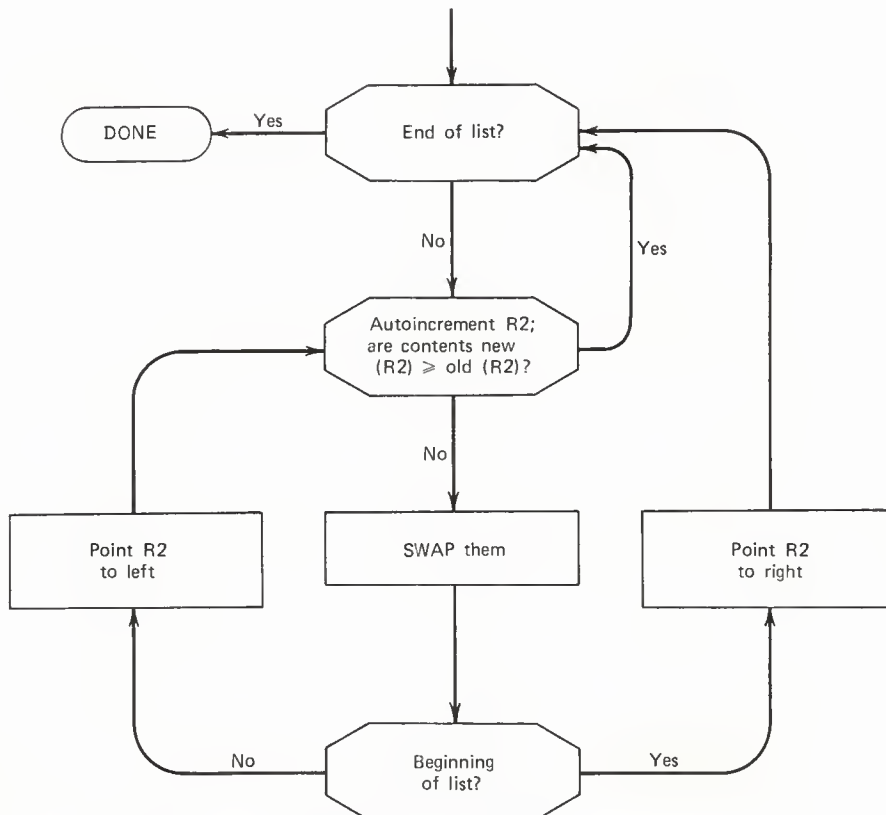


Figure 2.5 A flowchart to accompany the program in Figure 2.4

- EXERCISES:**
- (i) Write a complete program to accept input of a collection of numbers, sort them, and print them out in increasing order.
 - (ii) Data are stored in locations starting at **MEM**. The number of locations is given by the contents of register **R**. Write routines to:
 - (a) Delete from storage all null items (that is, when the location contains zero), by moving subsequent data down to replace them. Of course, the order of nonnull data items may not be changed, nor may their number be increased by both moving an item down and leaving it in its old location.
 - (b) Set to zero all data found after any zero data item.
 - (iii) Write a program to read two numbers and print out their quotient to one hundred decimal places, properly rounded. (*Hint*: "Teach" the computer grade school long division.)
 - *(iv) Write a program to read two numbers and print out the period of the decimal expansion of their quotient (for example, $3/7 = 0.428571$ with period 6).

2.4 WORD FORMAT

We discovered some time ago that in the instruction

INC Rn

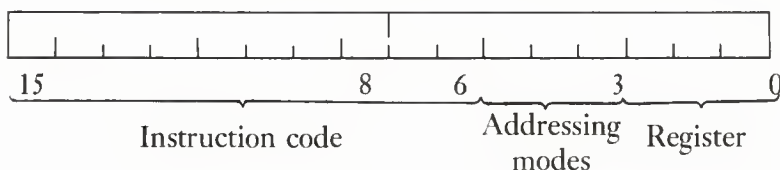
where **Rn** is one of the registers, the reference to **Rn** is encoded into the rightmost part of the word. Thus, a listing will show for **INC R4** the code **005204**. This is octal code, and the rightmost digit always, as you can easily check, gives the number of the register. The leftmost four digits, **0052**, are the coding of the instruction **INC**.

Remember that octal notation merely gives us a convenient way of representing the state of a 16-bit PDP-11 word (bit numbers are always octal). The rightmost octal digit represents the rightmost three bits, bits 0 to 2. The next four octal digits represent successively bits 3 to 5, 6 to 8, 9 to 11, and 12 to 14. Only one bit is left to be represented by the sixth and last octal digit representing the state of a PDP-11 word; this digit must therefore be either 0 or 1. Thus **0052**, the code for **INC**, is actually a ten-bit code.

This leaves one octal digit, whose general function may be discerned from a listing of instructions addressing a register in the various ways discussed in Section 2.3.

005201	INC	R1
005211	INC	(R1)
005221	INC	(R1) +
005241	INC	-(R1)

In general, we have the following word format for instructions such as **INC**, **DEC**, **NEG**, **CLR**, and **TST** that refer to just one location, register, or memory (*single-operand* instructions)



For convenience, our word format illustrations indicate octal digit boundaries and the half-word point.

In the encoding of a single-operand instruction, bits 3 to 5, represented by the second octal digit from the right, give the *addressing mode*. Each of the eight possible values yields a different mode, making the PDP-11 unusually rich in its facilities for accessing data. Each mode has a name; we have so far encountered the following ones.

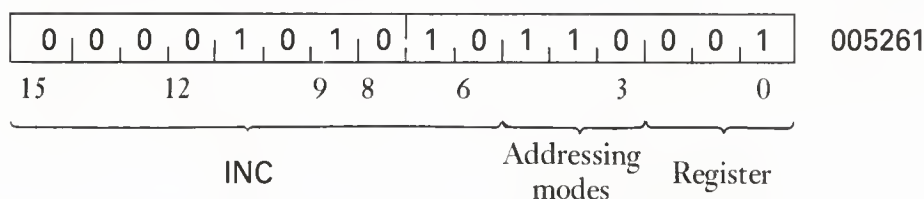
Mode number	Binary code	Name	Assembler syntax
0	000	Register	Rn
1	001	Register deferred	(Rn)
2	010	Autoincrement	$(Rn) +$
4	100	Autodecrement	$-(Rn)$

Mode 6 is called the *index mode*; the instruction

INC MEM(R1)

adds 1 to the contents of the address given by adding the *contents* of R1 to the *address* MEM. For example, if MEM has been loaded at location 1376 and R1 contains 100 (all octal), then MEM(R1) yields address 1476. This calculation of the address to be referenced by the instruction (the *effective address* calculation) is carried out in the CPU's internal registers and has no effect on the contents of R1 or MEM.

Since R1 is the register and the addressing mode is 6, the assembler will produce the word



We might regard this as the code for INC ?(R1) but, since every bit in the word has been used, there is no room to include the information that ? is to be MEM. The assembler will put the reference to MEM into the next word of the encoded program. Thus, on the PDP-11,

A single assembler language instruction may occupy more than one word of code.

This fact may already be familiar to you from your study of program listings. Note that listing programs designed for a wide printing output format will print all the code words for a given instruction on the same line as the instruction. In our example, the second word of the instruction INC MEM(R1) would be assembled with the *relocatable* address of MEM. If this were, say, 376, a program listing might show

005261	INC	MEM(R1)
000376'		

where we have omitted on the left the columns displaying the line of the program and the relocatable address at which the instruction itself is to be found.

Let us follow through what happens when the CPU encounters this instruction. Recognizing 0052 as the single-operand instruction INC, it proceeds to use the code 61 as the basis for its effective address calculation to locate the single operand. The 1 indicates register R1; the 6 indicates index

mode. Index mode prompts the CPU to place the contents of **R1** in an internal register. It then increments **PC** so that it points to the next word, fetches the contents of that word, and adds them to the contents of its internal register. The **INC** instruction is now performed, using the contents of the internal register as the address where the incrementing is to take place. **PC** is again incremented and now points to the first word of the next instruction.

EXERCISE: Why does the CPU not add the relocation constant to the word referring to **MEM**?

In its effective address calculation the CPU, in determining the mode, first examines bits 4 and 5 of the word. These cause it to treat the contents of the register selected by bits 0 to 2 as follows.

Bits		Use contents of register	Assembler syntax
5	4		
0	0	as address	Rn
0	1	as pointer, then increment	(Rn) +
1	0	decrement, then as pointer	-(Rn)
1	1	as index to contents of next word	MEM(Rn)

Note that no bit pattern allows automatic stepping to be combined with indexing, so forms such as **MEM(R1) +** are not permitted.

With the information obtained from bits 4 and 5 and the register reference given by bits 0 to 2, the CPU calculates an address in the manner just described. It then examines bit 3, the *indirect bit* in a single-operand instruction. If bit 3 is clear, the CPU treats the address it has found as the effective address and performs the instruction on its contents. This describes the action in modes 0 (register), 2 (autoincrement), 4 (autodecrement), and 6 (index).

If bit 3 is set to 1, however, the address found so far is not the effective address. Rather, it is treated by the CPU as a *pointer* to the effective address. In other words, the CPU replaces the address it has so far found with the *contents* of that address; those contents are treated as the effective address.

Suppose, for example, that the effective address calculation for an **INC** instruction yields **1376** before taking the state of bit 3 into account. Let us also suppose that the current contents of location **1376** are **2000**. If bit 3 is clear, then the effective address is, indeed, **1376**, and so location **1376** has its contents incremented, in this case from **2000** to **2001**. If, however, bit 3 is set, then **1376** is not the effective address. Rather, the contents of location **1376** form the effective address; that is, the effective address is location **2000**. Thus the contents of location **2000** are incremented in this case, and location **1376** is unaffected.

EXERCISE: Have we seen any examples of this indirect (or *deferred*) addressing so far?

The assembler recognizes the symbol **@** as indicating indirect addressing. Thus modes 1, 3, 5, and 7 would respectively use the syntax **@Rn**, **@(Rn) +**, **@-(Rn)**, and **@MEM(Rn)**. In mode 1 the assembler accepts **(Rn)** in place of **@Rn**, as we have already seen.

Reasonably enough, the more the CPU has to do in performing an instruction, the longer it will take to do it. Mode 0, in which the register specified is itself the effective address, is fastest. Among the direct modes, it is slower to index than to autoincrement or autodecrement; furthermore, index mode carries the overhead of assembling and loading an extra word of code. Indirect addressing takes the extra time necessary to fetch data from memory; register-deferred mode (mode 1), however, is as fast as automatic stepping. It is good programming practice to take execution times and program length into account. For example, if the instruction **INC MEM(R1)** occurs in a loop, it may well be better to **ADD #MEM,R1** before entering the loop, and **INC (R1)** in it. We shall see later that **ADD #MEM,R1**

takes up two words, so we would add one word to the program length. But if the loop is to be performed many times, this change to a faster instruction within it will be worthwhile.

EXERCISES:

- (i) What is the difference between INC (R1) and INC 0(R1)? Which is better?
- (ii) A block of words starting at location **TABLE** contains pointers to words holding the addresses of locations holding data. The number of words in the block at **TABLE** is in a word pointed to by **MEM**. Write a routine to clear all the data.
- (ii) Consider the following routine.

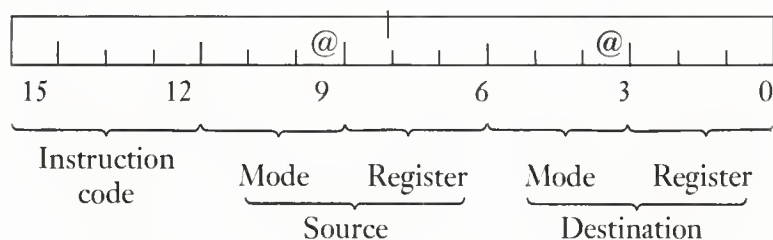
```

MOV      #MEM,R1
CLR      R0
LOOP:    CLR      @0(R1)
         ADD      #2,(R1)
         INC      R0
         CMP      R0,1000
         BNE      LOOP

```

- What does it do?
- Do we *need* to have `CLR @0(R1)` instead of `CLR @(R1)`?
- Would the routine be improved by replacing the first two lines of `LOOP` with the single instruction `CLR @(R1)+`?
- How could the routine be improved?

Double-Operand Instructions We learned from our investigation of single-operand instructions that six bits are needed to encode a reference to an operand: three to identify a register, three more for the mode in which it is to be addressed. Instructions such as **ADD**, **SUB**, **CMP**, and **MOV**, which refer to two locations (*double-operand* instructions), thus use up twelve bits for the operands. Such instructions, therefore, must have a four-bit instruction code. The first location referenced in the assembler language statement of these instructions is called the *source*, the second the *destination*; this is so even with the **CMP** instruction, which moves no data. The word format for a double-operand instruction is

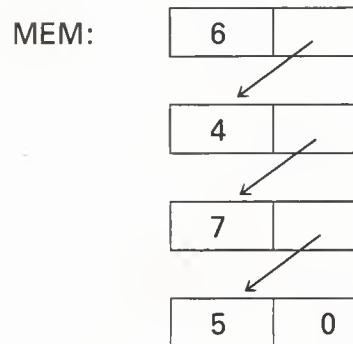


Either operand may be addressed in mode 6 (index) or 7 (index deferred), as in

ADD MEM(R1),@WRD(R2)

In this case the first code word would be **066172**, and the references to **MEM** and **WRD** would follow. (What is the instruction code for **ADD**?) The source memory reference, if there is one, always precedes the destination memory reference. So we see that double-operand instructions can generate one, two, or three words of code.

- EXERCISES:**
- (i) Can you suggest an improvement over `SUB #2,R1` (this instruction assembles as two words)?
 - (ii) Data are stored in memory in the form of a *linked list*: a number of two-word blocks scattered through memory, in which the first word holds data, and the second word points to the first word in the next two-word block. `MEM` points to the first word of the first block; the last block is identified by having its second word zero.
 - (a) Write a routine to store the data in consecutive locations starting at `WRD`.
 - *(b) Leaving all data items where they started in the linked list, store in the block at `WRD` the positions within the linked list of the data items in ascending order of magnitude. For example, if the linked list were



the block at `WRD` should contain 2, 4, 1, 3, since the smallest item is at the second position in the list, and so on.

Separate effective address calculations are performed for source and destination. As long as the two operands specify different registers, the choice of addressing mode in each is independent of the other.

There is also no problem in using the same register with both operands if neither addressing mode can alter the contents of the register. With a closer awareness of the CPU's order of operations, we can even allow one or both addressing modes to increment or decrement the common register. The CPU goes through the following sequence for each instruction.

1. Fetch first word of instruction from memory.
2. Increment `PC`.
3. Calculate source effective address (incrementing `PC` in the process for mode 6 or 7).
4. Perform any automatic stepping of source register.
5. Calculate destination effective address (incrementing `PC` in the process for mode 6 or 7).
6. Perform any automatic stepping of destination register.
7. Perform the instruction, using addresses found in Steps 3 and 5. (*Proviso*. But with original contents of *source* address.)

Note that the addressing mode used for the source can influence the effective address calculated for the destination, since the contents of the register are stepped before the destination effective address is calculated. Consider, for example,

`MOV       (R1)+,(R1)`

Suppose that when this instruction is reached, `R1` contains 2000, location 2000 contains 10, and location 2002 is clear. At Step 3 the source effective address is computed as 2000. At Step 4 the

contents of **R1** are set to **2002**; consequently, at Step 5, the destination effective address is calculated as **2002**. Performing the instruction leaves **10** in both locations **2000** and **2002**, and **2002** in **R1**.

The source addressing mode can also affect the *contents* of the destination effective address rather than the effective address calculation itself. Consider

ADD (R1)+,R1

with the same initial contents as before: **2000** for **R1** and **10** for location **2000**. The source effective address is **2000**. After Step 4, **R1** contains **2002**; the **ADD** instruction adds the contents of location **2000** to this, so that finally **R1** contains **2012**.

- EXERCISES:**
- (i) Can the choice of source addressing mode in an instruction affect both the calculation of the destination effective address and the contents of the address eventually calculated?
 - (ii) Can you improve the sorting routine of Section 2.3?
 - (iii) Can the result of **ADD #4,R1** be achieved more efficiently?

The CPU's order of operations is such that under no circumstances can the addressing mode used for the destination affect the effective address calculation for the source. The order of operations does not in itself, however, protect the *contents* of the source effective address when the source is addressed in register mode. Consider, with the same initial contents as before,

ADD R1,(R1)+

Step 5 calculates the destination effective address as **2000**; Step 6 increments the contents of **R1**, making them equal to **2002**. Without the proviso in the description of Step 7, the instruction would now add **2002** to the contents of location of **2000**, resulting in $10 + 2002 = 2012$. Some, but not all, models of CPU for the PDP-11 do treat such instructions in just this fashion. Most models of CPU, however, implement the proviso noted in Step 7. In our example, although **R1** has been autoincremented at Step 6 and contains **2002**, its original contents, **2000**, would be added to location **2000**, resulting in **2010**.

It is usually best to avoid instructions that are not compatible among all members of the PDP-11 family. If such an instruction does happen to be particularly convenient, be sure to include adequate comment to guard against problems should the program ever be used with another CPU. The assembler will output an error message when it compiles such an instruction, but this does not generally prevent your using it. In the program listing a **Z** will appear against the instruction as a warning that it is not compatible among all CPUs. This is one of a variety of codes indicating an error detected by the assembler. Of course, the assembler can only reveal your abuse of the rules of assembler language (*syntax* errors); it cannot warn you that your program is not structured to do what you intended it should (*logical* errors). You have doubtless already seen in your program listings several of the following codes.

A	Address within instruction incorrect
E	No .END statement; most assemblers will provide one
M	Same label used more than once
U	Undefined symbol

- EXERCISES:**
- (i) Discuss the effect of **ADD R1,-(R1)** with and without the Step 7 proviso.
 - (ii) Determine how your own system treats these instructions.

Program Counter Addressing It remains to discuss the addressing modes in instructions of the form **CMP #100,R1** and **CLR MEM**, in each of which an effective address calculation does not apparently reference a register. The simplicity of these assembler language statements conceals a degree of subtlety in the code into which the assembler translates them.

An operand that is just a number, such as the source in **CMP #100,R1**, is said to be addressed in *immediate mode*. The assembler encodes this instruction into two words, each of which is individually expressible as an assembler language statement.

022701	CMP	(PC)+,R1
000100	.WORD	100

The second word just holds the operand, **O 100**. In the first word, **PC** is the program counter. On the PDP-11 the program counter is one of the general registers accessible to the user. It is, in fact, register **7** and if, as is quite likely, your assembler does not recognize the conventional register mnemonics, your program should include the assignment statement

PC = %7

Thus, when broken down into its constituent parts, immediate mode addressing appears as autoincrement addressing referencing **PC**, followed by a statement of the data. The only reasons not to use the two-line form of the code are that it is less suggestive of the purpose of the code and that it takes longer to type.

It is important to understand exactly how immediate mode addressing works. Let us suppose that the two-line code for **CMP #100,MEM** is loaded into locations **1100** and **1102** (remember that word addresses are even). The previous instruction will have set **PC** to contain **1100**. The CPU now fetches the contents of the location pointed to by **PC**; that is, it fetches the instruction **CMP (PC)+,R1**. Then, *before proceeding to execute that instruction*, the CPU updates **PC** to point to the next word; so **PC** now contains **1102**. It is crucial to bear in mind that **PC** is incremented (by **2**, of course) before the CPU begins the calculations required by the current instruction word.

Thus the CPU determines the effective address for the source as being location **1102**; so the **CMP** instruction will compare the contents of location **1102** (that is, **100**) with the contents of register **R1**. This is the comparison we wanted.

Before making the comparison (but, as we saw above, after completing the effective address calculation for the source), the CPU, *in response to the use of autoincrement addressing*, increments the contents of **PC** by **2**. **PC** now contains **1104**, the address of the first word of the next instruction. The CPU fetches that word, increments **PC**, and so program execution continues.

The same effect could be achieved, less economically, by clearing **R0**, then instructing

026001	CMP	#00(R0),R1
000100		

At first sight it may seem puzzling that this instruction increments **PC** by a total of **4** without any specific autoincrement requirement in the assembled code. Proper incrementing of **PC** is, however, built into the workings of index mode. The CPU increments **PC** in the usual way when it fetches the first word of the **CMP** instruction, so that it points to the second word of the instruction, containing the data to be indexed. Index mode now requires the CPU to fetch that word, *increment PC*, and perform the **CMP** instruction.

On the other hand, **CMP #100,R1** results in a *one-word CMP* instruction in which the source effective address has been cunningly arranged to be the next word. It works because the assembler, in response to the syntax of the instruction, has placed the data to be indexed in that next word. One-

word instructions increment PC by 2 once only, so autoincrementing has to be used to carry PC past the word containing data. This is another example of our old concern to ensure that the CPU does not try to execute a word of data as if it were an instruction.

Immediate mode may, as we have seen in instructions such as `MOV #MEM,R1`, reference relocatable data. Assembly is just as discussed previously, with the relocatable value housed in the word following the instruction and awaiting adjustment at load time.

- EXERCISES:**
- (i) What is the effect of `CLR #MEM`?
 - (ii) Does `INC #100` give 101 as its result?
 - (iii) What is the effect of the following sequences?
 - (a) `INC (PC)`
`DEC (PC)`
 - (b) `INC (PC) +`
`DEC -(PC)`

Relative Mode The normal way to reference memory in PDP-11 assembler language is by direct expression of names assigned by the programmer to memory locations, as in `CLR MEM` or `MOV MEM,WRD`. This form of referencing memory is called *relative mode* addressing, and it also uses PC as its register. The instruction `CLR MEM` is interpreted by the assembler as `CLR X(PC)`, where X is a number calculated by the assembler. Following through an example will show us how the assembler must calculate X to make the instruction work. Let us suppose that the first word of the `CLR MEM` instruction assembles into location relocatable 100 and that `MEM` assembles into relocatable 376. When PC is used in the effective address calculation, this `CLR` instruction in index mode has incremented the contents of PC *twice*, as previously described; thus this calculation is performed with PC containing relocatable 104. The quantity X must be such that when added to 104 it yields 376; so X must be 272, and the instruction `CLR MEM` assembles as if it were `CLR 272(PC)`.

000100	005067		CLR	MEM
000102	000272			
...				
...				
000376	000000	MEM:	.WORD	0

In general, X must be equal to

$$\text{address of MEM} - \text{address of word after location of } X$$

because, when the CPU performs the effective address calculation, PC contains the address of the word after that in which X is stored. X may be stored in the second or third word of the instruction, depending on which operand is being addressed in relative mode; the assembler will automatically make the necessary adjustments.

This discussion has proceeded in terms of relocatable addresses, although the CPU is, of course, operating with load addresses. In fact, this makes no difference, which is just as well because the assembler, if it is given no idea where a program will be loaded, can only supply relocatable addresses. It makes no difference because, as we have just seen, the assembler determines what to put in the second word of an instruction such as `CLR MEM` by subtracting one program address from another, forming the *relative* address of the two words that gives this addressing mode its name. The word distance between two addresses will not be changed if both are relocated in the same amount by the addition of a common relocation constant. Note that this distance is itself *absolute*, not relocatable.

EXERCISE: Encode the following program, with **START** loaded at address 1000, given that the branch instruction occupies one word of code.

```

START:    INC      MEM
          INC      MEM
          ADD      #1, MEM
          BR       START
MEM:      .WORD    0
          .END     START

```

The code assembled for relative mode instructions is an example of *position-independent code* (PIC). This may seem a curious designation since, as you saw in the last exercise, the code depends on the position of the instruction in the program. It is, however, independent of where the program is loaded into memory; as we observed, the difference between two program addresses is unchanged when the entire program is relocated. In timesharing systems there is an economy in writing systems programs entirely in position-independent code. These programs are shared by many users; it would be wasteful to juggle each user's memory space to give such programs the same load address each time or to relocate position-dependent code for different load addresses. Similar considerations apply, in any system, to a program that will be loaded after user programs of varying lengths; debugging programs are of this kind.

Negative Numbers This section ends with a brief discussion of the representation of negative numbers in PDP-11 assembler language and its coding.

In assembler language statements, negative numbers are represented by a $-$ sign in the usual way. For example,

```
MOV      #-1, -2(R1)
```

or

```
MEM:     .WORD    -1
```

Note that $-2(R1)$ indicates the number -2 , indexed by the contents of **R1** (*not* by the negative of those contents). It is a convenient way of stepping back one word from the one pointed to by **R1**, without affecting **R1**.

Negative numbers may, of course, also result from a calculation. The representation of a negative number within a PDP-11 word does not depend on how it came to be there. Bit 15 of the word is called the *sign bit* and is set to 1 for negative numbers. However, as you can easily check for yourself (how?), the representation of, say, -1 is far from being that of 1, save only for a 1 rather than a 0 in bit 15. The architecture of the PDP-11 uses the convention known as *twos complement* to represent negative numbers. If X is a positive number, to form the representation of $-X$:

1. Form the representation of X ; since X is positive, the sign bit will be clear, and bits 0 to 14 will contain the binary code for X .
2. Subtract 1.
3. Change all 0s to 1s and all 1s to 0s.

With this convention, the largest positive number that can be stored in a PDP-11 word has 0 in bit 15 and 1s everywhere else. Its octal code is 077777, and its decimal value is $2^{15} - 1 = 32\,767$.

Observe that -1 is represented by 1s in all bits: 177777; -2 is 177776; and so on. We have the following spectrum of values.

Positive	Largest	077777
		077776
	Smallest	000001
		000000
Negative	Largest	177777
		177776
	Smallest	100001
		100000

Note that in accordance with proper usage, we describe -2 as smaller than -1 , and so forth.

The suggestion of an automobile mileage indicator run backward may make this table of representations seem a little more natural. In any case, familiarity will soon make it less mysterious.

- EXERCISES:**
- (i) How is the negative of **O 100000** represented in a PDP-11 word?
 - (ii) Describe a systematic method for obtaining the *octal* code representing the negative of the positive number **X**.
 - (iii) What is the effect of

```
CMP      #-77777,#77777
BMI      MINUS
```

What do you suppose is the problem?

- (iv) If **R0** contains **77777**, what is the effect of **INC R0**?
- (v) The label **MEM** is attached to location relocatable **100**. How would the instruction **CLR MEM** be coded if it were assembled starting at location relocatable **376**?

3 Program Structure

3.1 SUBROUTINES

In Chapter 2 we discussed the block structure of our programs. The general approach was to divide the problem to be tackled into small sections. For each section we then write a routine, and the complete program is made up of the collection of routines together with various connections between them. When the program is executed, it proceeds through the various routines in some order. Typically, the order has been: input, calculation, output.

More complicated problems, however, may lead to programs that *branch*. That is, there may be various routines leading from or to any given routine. For example, we might want to have results printed out at different stages of execution. We cannot readily do this using just one printout routine, because when printout is finished, the program has lost track of the point it had reached before branching; so it cannot pick up the calculation where it left off. The crux of the matter is that our printout routine is, once written, a fixed entity of the program. We can jump to it from as many points as we like; but there is only one way to leave it, that which is written into the program.

It is true that we could terminate the printout routine with conditional branch instructions, and manipulate these to set us back to the point from which we branched. But this would be very complex and cumbersome. It would also be pointless, since we have at our disposal the possibility of creating a *subroutine*, which is designed exactly with this difficulty in mind.

Before dealing with how to write subroutines, let us consider their effects. Suppose that our printout routine has been written as a subroutine, with its first line labeled **PRINT**. Printout is achieved by transferring program control to location **PRINT** while taking steps to ensure that after printout the program can carry on from where it left off; this process is described as *calling* the subroutine.

Using subroutines, the approach to complex programming tasks is greatly simplified. In our initial sketch of a program, we would not trouble to consider the mechanics of, for example, a printout routine. If at some stage we have a quantity in location **MEM**, which we want printed out as a decimal number, we might write simply

PRINT MEM

in our first draft, just as if there were the single assembler language instruction **PRINT** to print out the number for us. Regrettably, there is no such instruction so, before running the program, we would replace this line by a *calling sequence* to location **PRINT** and write the appropriate subroutine.

This is a good approach whenever some task must be carried out many times. To begin with, we suppose that there is a single instruction to accomplish the task. Then, when the structure of the program has been worked out, it is time to fill in the necessary subroutines.

A Text-Editing Program We shall approach the problem of writing a subroutine in the context of a program to edit text stored in memory. Suppose that we want to allow only one space between words, so that our program must suppress any superfluous spaces. We can do this by replacing the ASCII code for space with 0 and subsequently ignoring null words in the block holding our text.

We can do this easily enough without writing a subroutine. If R2 is being used as text pointer, the following code will serve our purposes.

```

                                CMP      #40,(R2) +           ;space encountered?
                                BNE      ONWARD              ;no—go on processing
SPACE:                         CMP      #40,(R2)             ;yes—suppress extra ones
                                BNE      ONWARD
                                CLR      (R2) +
                                BR       SPACE
ONWARD: ...
```

This code could just be included in the main routine that steps its way through our text. Unfortunately, it does not deal with all the circumstances that arise during text processing when a space is encountered. For example, after a period we might want to allow two spaces, suppressing all extra spaces after the second one. And after ↵ we might want to let one or more spaces indicate a desire for a paragraph and have the program supply a fixed indentation of, say, four spaces. Of course, all these options could be handled by suitable branch instructions preceding our space-suppressing code. But the result would be a long, unbroken sequence of code, so entangled with branch instructions that its structure would be totally obscured. Indeed, it is likely that the presence of such coding would indicate a certain lack of thought about the structure of the program. In any case, labyrinthine coding encourages errors and makes them harder to detect, is frustrating to read, and can be a barrier to further development of the program.

It is also wasteful to bury the space-suppressing routine in the larger routine that responds to encountering a space in the text. We may want to suppress space at other times, such as preceding a ↵ or punctuation mark. There is surely no reason to repeat the coding in each place.

These considerations all point to isolating the space-suppressing routine so that it may be called on whenever it is needed but does not clutter the development of a well-structured program. Thus we would include in our program a self-contained block of code such as

```

;subroutine to suppress spaces from (R2) on
SPACE:    CMP      #40,(R2)
          BNE      RETURN
          CLR      (R2) +
          BR       SPACE
RETURN: return to main program
```

We must write the subroutine so that the return is to the next instruction after the procedure that called the subroutine. This is inherent in the concept of a subroutine. Any decisions as to what to do next are to be made by the main program after the subroutine has returned control. The whole notion of breaking down a programming task into separately coded components, and expressing the logical structure of the task by a chart of transfers of control between the various components, positively requires that one component refrain from meddling in the job assigned to another.

Thus the return must be to the location given by the contents of PC when control was transferred to SPACE (remember that PC always points to the *next* instruction). Thus the calling sequence for the SPACE subroutine—and, indeed, for any subroutine—must save the contents of PC.

An obvious place to save **PC** is in another register; let us suppose that **PC** has been saved in **R5**, so that the return should be made to location (**R5**). We cannot, however, accomplish this by **BR (R5)**; this form is meaningless, because the destination in a branch instruction must be one that the *assembler* can calculate, subject only to relocation. The loaded code in a branch instruction must give the actual destination address; there is no effective-address calculation.

We can, however, transfer control to the location pointed to by **R5** in a very simple manner. All we need to do is make **PC** point to the same location, which is achieved by

```
MOV      R5,PC
```

With this single instruction as the return, we have a complete subroutine.

The calling procedure for the subroutine must now be investigated. We need something like

```
MOV      PC,R5
BR       SPACE
```

although this would not quite fulfill our needs (why not?). There is, however, a special instruction that saves **PC** in a specified register and transfers control to a specified destination in a single operation. This is the Jump to SubRoutine instruction **JSR**, and here we would use

```
JSR      R5,SPACE
```

The **JSR** instruction is encoded, reading from left to right of the word, as the seven-bit instruction code **004**, three bits for the register specified to store **PC**, and six bits for the destination. A full effective address calculation is carried out for the destination. (Does it save *space* to use a **JSR** instead of saving **PC** and using a branch instruction?) Note, however, that for saving **PC** a *register* must be specified. A **JSR** instruction saves the current value of **PC**, which therefore points to the instruction following the **JSR**, as is required.

If our program is going to do nothing but suppress these superfluous spaces, the main part of it would now be as follows.

```
LOOP:    CMP      #40,(R2) +
          BNE      LOOP
          JSR      R5,SPACE
          BR       LOOP
```

Our subroutine returns straight into a branch instruction; it may be tempting to adjust the return to point back to **LOOP** and so save an instruction. But, for the reasons discussed earlier, this would be false economy. The subroutine has an independent task to perform and should not help the main program to do something else. Furthermore, we may later decide that instead of branching straight back to **LOOP** we want to perform other text-editing functions. In that case we can just extend the main program without having to amend the subroutine. Note that it will not then be necessary to branch back to **LOOP** after the return from **SPACE** in order to check whether the current character is a space (why not?).

Even in such a trivial program we must take scrupulous care that the main program leaves things *exactly* where the subroutine expects to find them. The subroutine will clear spaces from (**R2**) onward, returning when it encounters some other character. However, when the main program finds a space in (**R2**), it can only be the first space after some other character (why?) and, therefore, should be kept. Hence, when the main program finds a space, it autoincrements **R2** before calling **SPACE**, so that the subroutine does not delete the first space.

EXERCISE: Build from our main program and subroutine a program to read a line of text from the terminal into memory and print it out with superfluous spaces deleted. Annotate your program and draw a flowchart for it.

Nesting Subroutines Let us develop our text-editing program to deal with periods. We can amend our main program even before deciding what we are going to do with periods. After dealing with the issue of whether the current character is a space or not, instead of branching back to **LOOP** we want to go on and check whether it is a period. So we will change our four-line main program to

```

LOOP:      CMP      #40,(R2)
           BNE      L1
           TST      (R2)+
           JSR      R5,SPACE
L1:        CMP      #56,(R2)

```

This time we have refrained from autoincrementing at **LOOP**, because then we would have had to compensate by autodecrementing at **L1** (why?), and the program would have become confusing. This cost us an extra instruction, but it is important to keep a clear idea of where **R2** is pointing. Anyway, there is the compensation of avoiding a pointless check when leaving **SPACE** (what check?).

Now we can complete the main program with

```

           BNE      L2
           JSR      R4,PERIOD
L2:        TST      (R2)+
           BR       LOOP

```

This makes a few assumptions about subroutine **PERIOD**. It must expect to find **R2** pointing to a period when it is called. It must return with **R2** pointing to the character preceding the next one to be checked at **LOOP**. And it must return via **R4**. Of course, we are not permanently bound by these assumptions. If we find it preferable to enter or leave **PERIOD** differently, we can always make appropriate changes in the main program. The goal of a nicely matched main program and subroutine is not always reached at the first attempt.

Subroutine **PERIOD** will merely delete a space if one was typed before the period and will allow just two spaces after the period. For simplicity the program requires that at least two spaces be typed after a period; the subroutine deletes any surplus over this.

To delete a possible space before the period, **PERIOD** need only decrement **R2** and call **SPACE**; it returns with **R2** again pointing to the period. Next, **PERIOD** must move **R2** on past the two spaces allowed and call **SPACE**. Before returning, **R2** must be decremented to accord with the expectations of the main program.

```

PERIOD:    CMP      #40,-(R2)
           BNE      P1
           JSR      R5,SPACE
           TST      -(R2)
P1:        ADD      #10,R2
           JSR      R5,SPACE
           TST      -(R2)
           MOV      R4,PC

```

You may object that we should not call **SPACE** to delete a single space when **CLR (R2)** would suffice.

However, we may later want to develop the program so that instead of leaving null words when we delete, we actually move all subsequent data items down one location. We prepare for such a development by using **SPACE** for every deletion, so that later we need only amend this subroutine.

A flowchart for the whole program is shown in Figure 3.1. The symbols **+** and **-** indicate pointing **R2** forward or back one word.

- EXERCISES:**
- (i) What most necessary provision is missing from our flowchart and program?
 - (ii) Suppose that several spaces were typed before a period. **SPACE** will delete all but one of them; will the first three lines of **PERIOD** delete the remaining one?
 - (iii) Write the complete program. Now rework it completely so that subroutines and main program fit together as smoothly as possible.

Our program features the phenomenon of one subroutine calling another. This is called *nesting* of subroutines. A subroutine may be nested within a nested subroutine, and so on. It is by no means uncommon to find six subroutines, each nested within its calling subroutine; the *depth*, or *level*, of nesting is then six.

It would seem, however, that the depth of nesting would be limited by the number of registers available. We used **R4** to call **PRINT** on the grounds that, when **PRINT** calls **SPACE**, **R5** is needed for the return **PC**; therefore **PRINT** must use another register. It is so crucial to the development of programs for complex tasks that subroutines can be nested to any depth that having to use a different register for each level of nesting would be a most severe limitation on the powers of the PDP-11. However, unlimited nesting is available, because the **JSR** instruction does more than we have indicated so far.

Linkage Register PDP-11 hardware is designed for efficient and effortless transfer of control to and from subroutines. When subroutine calling and return sequences are considered together with the hardware features that underlie them, the whole process is called the subroutine *linkage*. The register referenced in a **JSR** instruction is called the *linkage register* or *linkage pointer*. We have seen that imposing a limit on depth of nesting is only avoidable if the same linkage register can be used at different levels. This is possible on the PDP-11 because, in addition to its other functions, the **JSR** instruction *saves the original contents of the linkage register*. Thus, the single instruction

JSR R5,SUB

1. Saves the contents of **R5**.
2. Moves the contents of **PC** into **R5**.
3. Moves *address* **SUB** into **PC**.

Discussion of *where* **JSR** saves the contents of the linkage register must wait until Section 3.2.

To see the effect of this, let us suppose that in our text-editing program both **SPACE** and **PERIOD** use **R5** as linkage register. When the main program calls **PERIOD** with

JSR R5,PERIOD

R5 is set to hold the return **PC**, with its original contents saved elsewhere. Now subroutine **PERIOD** calls subroutine **SPACE** with

JSR R5,SPACE

This puts the **PC** for the “**SPACE** → **PERIOD**” return into **R5**. It also *saves the former contents of **R5***; that is, it saves the “**PERIOD** → main program” return **PC**.

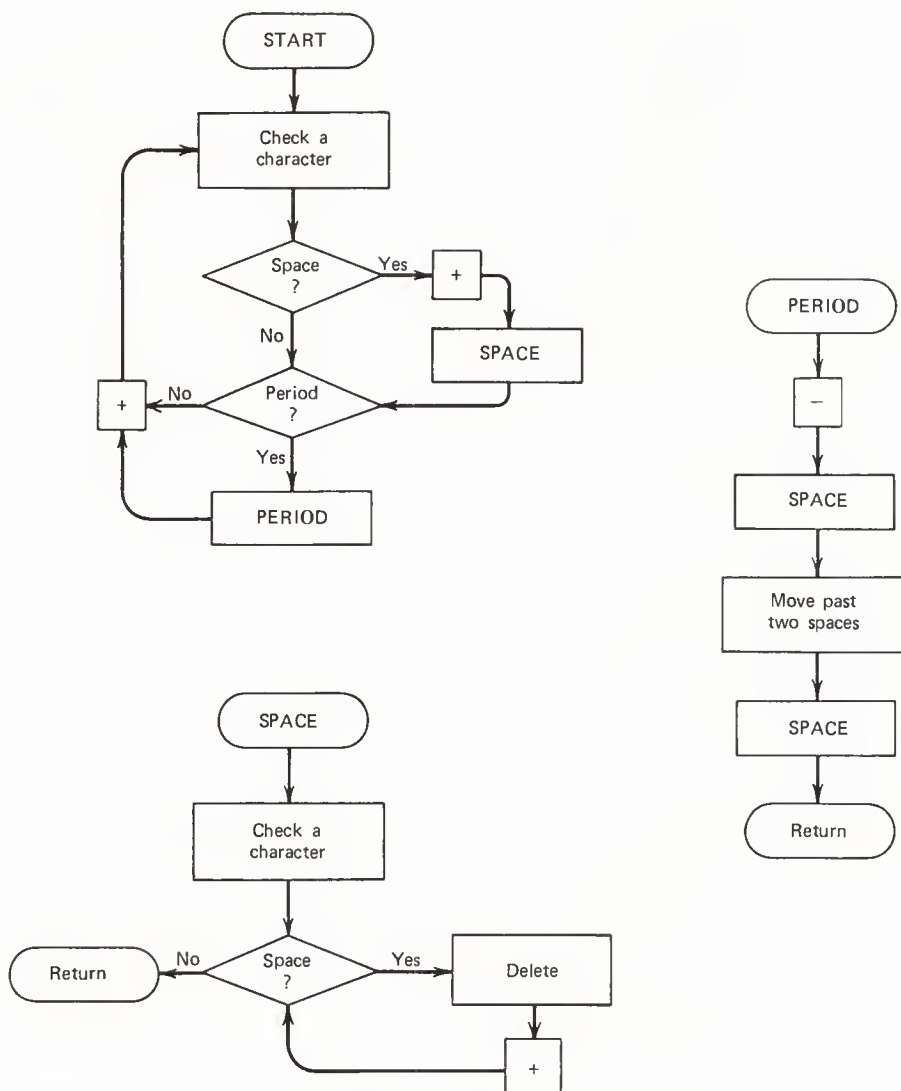


Figure 3.1 A flowchart for the text-editing program.

It is one thing to know that the information needed for both subroutine returns has been preserved, but it is another matter to get at it. We can still return from **SPACE** to **PERIOD** with **MOV R5,PC**; but since we do not know where **JSR R5,SPACE** saved the **PC** for the “**PERIOD** → main program” return, we cannot manage the return to the main program.

The trouble is that on entering subroutine **SPACE** we saved certain information, without *restoring* that information when we returned from **SPACE**. Instead of **MOV R5,PC**, we must return with the special **ReTurn** from Subroutine instruction **RTS**. The instruction

RTS R5

1. Moves the contents of **R5** into **PC**.
2. Moves into **R5** the last address saved by a **JSR** instruction (and not already restored by a previous **RTS**).

Thus **RTS** not only restores control to the main program or to the subroutine at the next level in the nesting sequence, but also tidies up after the **JSR** that called the subroutine. The **JSR** – **RTS** pairing

offers remarkably smooth, essentially automatic subroutine linkage, leaving the programmer free to concentrate on the substance of the task in hand.

It is even feasible for the main program to use a linkage register for other purposes, knowing that it will be restored when the subroutine returns. But greater clarity results from using a register exclusively as a linkage register. **R5** is customarily reserved for this purpose. A most important consequence of the way in which **JSR** and **RTS** are paired is that:

One linkage register may be used for all subroutine calls and returns.

Suppose, for example, that subroutine **SUB1** calls **SUB2** which, in turn, calls **SUB3**. Thus we have the situation depicted in Figure 3.2. The three **JSR** instructions save the following three data items.

Return address "SUB2 → SUB1"
Return address "SUB1 → main program"
Original contents of R5

with the *bottom item being the one saved first*, then the middle one, finally the top one. The return address "SUB3 → SUB2" never gets saved, because subroutine **SUB3** does not issue a **JSR** to a deeper level of nesting.

It may be helpful to imagine successive **JSRs** placing the information they save neatly on the top of a "stack" of such items of information. It is in deference to this image that our list of items saved by **JSRs** puts the first one saved at the bottom of the stack.

The restorative function of **RTS** may now be seen as removing the top item from the stack and putting it into **R5**. Thus, referring again to Figure 3.2 and our stack, the first **RTS R5** to be performed comes from **SUB3**. The contents of **R5**, set by the **JSR** in **SUB2**, indicate a return to **SUB2**. In addition, this **RTS** removes the return address "SUB2 → SUB1" from the top of the stack and puts it into **R5**.

Subroutine **SUB2** now runs its course, concluding with **RTS R5**. The contents of **R5** (restored by the **RTS** in **SUB3**) indicate a return to **SUB1**. In addition, this **RTS** removes the top item of the stack, which is now the return address "SUB1 → main program", and puts it into **R5**.

Finally, when **SUB1** terminates, **RTS R5** returns control to the main program and restores in **R5** its original contents, the only item left on the stack by this stage.

Perhaps the most satisfying aspect of this system of subroutine linkage is that once it is properly understood there is rarely any need to focus one's attention on it. If **R5** has been chosen as linkage register for the program, *every* subroutine returns with **RTS R5**. And this is the case regardless of how complex the network of subroutine control transfers may be.

<u>Main program</u>		<u>SUB1</u>		<u>SUB2</u>		<u>SUB3</u>
...		SUB1:	...	SUB2:	...	SUB3:
...		...	...	...	...	...
...		JSR R5,SUB2	...	...	...	...
JSR R5,SUB1		...	...	...	...	...
...		...	...	JSR R5,SUB3	...	RTS R5
...		RTS R5	...	...	...	...
...			...	RTS R5	...	...
...						

Figure 3.2 A subroutine linkage.

- EXERCISES:**
- (i) Write subroutines to perform input and output of decimal numbers. Have your subroutines call on other subroutines to perform multiplication and division.
 - (ii) Write a subroutine to print out the numerical results of its main program in tabular form. Allow eight spaces per column, and right justify the columns (that is, the rightmost digits of all numbers in a column are to be vertically aligned). Your printout subroutine should string out the remainders in memory and then call a subroutine to count spaces and columns.
 - (iii) Write a program to print out a table of all prime numbers less than ten thousand.
 - *(iv) Write a program to read a PDP-11 assembler language instruction typed in at the terminal and print out its coding, assuming some fixed load location. Cope with as many types of instruction and addressing modes as you can.

ASCII Text Many programs other than text-editing programs need to print out ASCII text. Most often this is in the form of a message requesting input or reporting on the progress of the program. We now know how to write a subroutine to print out such a message, but there remains the problem of how to load the message into memory in the first place, since this is not text to be input at run time. We could use a `.WORD` directive, but this would be tedious. To load the message `WAITING FOR GODOT` into a block starting at `MEM`, we would need

```
MEM:      .WORD      127,101,111,...
```

and so on, for fourteen more characters. If there is not room for all of these to be typed on one line, one or more additional `.WORD` directives must be issued; an assembler language statement may not extend over more than one line.

Fortunately, there are assembler directives to ease our task. At the very least we can save ourselves the trouble of looking up or memorizing ASCII codes, because the assembler accepts the symbol `'` as indicating that the next symbol is to be interpreted as its ASCII code. Thus the preceding sequence could be replaced by

```
MEM:      .WORD      'W','A','I',...
```

and so on.

This is not the best solution but, before proceeding, it is worth noting that ASCII code generation using `'` is very useful in other contexts. Since `'0`, for example, is just the same as `60`, instead of

```
CMP      #60,(R1)
```

we may write

```
CMP      #'0,(R1)
```

saving the trouble of checking the ASCII code and making the program easier to read. This form should only be used with regular printing characters (never with `CONTROL` characters).

The easiest way to have the assembler store ASCII text in memory is to use one of the special directives `.ASCII` or `.ASCIZ`. The syntax is

```
MEM:      .ASCII      /WAITING FOR GODOT/
```

and similarly for `.ASCIZ`. Observe the *delimiters* `/ . . /` that introduce and terminate the text without being regarded as part of the text. Most printing characters may be used as delimiters; `/`, `"`, and `'` are

popular choices. The assembler will note which character was used as delimiter to introduce the text and go on assembling ASCII code until it encounters the same character again.

Note that `.ASCII` and `.ASCIZ` are neither assembler language instructions nor monitor calls. They are directives to the assembler program to encode in the manner indicated, similar to `.WORD` directives.

Neither of these directives may be followed by a message extending over more than one line in the assembler language program. However, since an individual character may be presented in an `.ASCII` or `.ASCIZ` directive by enclosing its ASCII code within angle brackets (*outside* the delimiters), we may store the text

```
WAITING
FOR GODOT
```

in memory starting at `MEM` by

```
MEM:      .ASCII      /WAITING/⟨15⟩⟨12⟩/FOR GODOT/
```

These directives store the characters in *bytes*; each seven-bit ASCII code occupies all but the highest bit of an eight-bit byte. The preceding message would appear in memory as

A	W	MEM
T	I	MEM + 2
N	I	MEM + 4
...		
...		

and so on.

There is no difficulty in writing a routine to print out a message stored in bytes. If we begin with `MOV #MEM,R1`,

```
.TTYOUT      (R1)
```

will print out the `W` at the terminal. The `A` in the high byte of `MEM` does not cause confusion, because the `.TTYOUT` monitor call works by moving a *byte* from the location specified into `R0` and then going into a monitor-controlled routine to output the character from `R0`. The `.TTYOUT` system macro for this call on it will include the code

```
MOVB          (R1),R0
```

using the instruction `MOVB` to MOVE a Byte. The source is the byte pointed to by `R1`, which is the low byte of `MEM`; remember that the address of a word is the same as that of its low byte. The destination is the low byte of `R0`; a byte instruction referencing a register in mode 0 always refers to the low byte of the register. The high byte of a register does not have an address; the assembler will interpret `R0 + 1` as meaning `R1`.

In order to print out the next character, `A`, `R1` must be pointed to the next byte, the high byte of `MEM`. Thus we can print out a string of characters stored in bytes by

```
LOOP:        .TTYOUT      (R1)
              INC          R1
              BR           LOOP
```

Note that we point R1 to the next byte by adding 1 to its contents. However, an easier way is

```
LOOP:      .TTYOUT    (R1)+
           BR         LOOP
```

The reference to R1 appears in the expansion of the system macro as

```
MOVB      (R1)+,R0
```

When the CPU carries out this instruction, it will move the data; then, *because MOV B is a "byte-oriented" instruction, it will increment R1 by 1 only.* We shall discuss byte-oriented instructions more fully in Section 3.3.

Another way of storing data in bytes uses the .BYTE directive, with syntax similar to that of the .WORD directive:

```
MEM:      .BYTE      'W','A','I',...
```

and so on. Another method uses the .WORD directive with the symbol " to indicate that the next *two* characters are to be interpreted as ASCII code and stored one in each byte of the word:

```
MEM:      .WORD      "WA","IT","IN",...
```

and so on.

We must have some way of marking the end of the message unless we want our printout routine to step through all of memory. An easy way is to append a null byte to the text and check for it with a conditional branch instruction.

```
LOOP:      MOV B      (R1)+,R0
           BEQ        FINIS
           .TTYOUT
           BR         LOOP
FINIS:     RTS        R5
```

Would

```
LOOP:      .TTYOUT    (R1)+
           BNE        LOOP
           RTS        R5
```

be satisfactory? (Answer this again after reading Chapter 4.)

The sole difference between .ASCII and .ASCIIZ is that the latter automatically appends a null byte to the text specified. This fits perfectly with our routine.

Instead of writing our own routine, we may be able to use a monitor call. The RT-11 monitor call is .PRINT. The call

```
.PRINT      #MEM                      ;note the syntax!
```

will print out bytes starting from MEM. On encountering a null byte, it will print a ↵ and then return control to the program. This fits with the .ASCIIZ directive. Do not forget that an .MCALL directive for .PRINT must be included in the program.

If we want to use `.PRINT` but do not want the terminating `↵`, we can finish our text with a byte containing 200; this causes `.PRINT` to return control without issuing a `↵`. To obtain this form, we use the `.ASCII` directive to store the text with no extra byte and then append our own extra one.

```
MEM:      .ASCII      /WAITING FOR GODOT/
          .BYTE       200
```

Since either directive may generate an odd number of bytes, before doing anything that will need to begin on a word boundary we must issue the directive `.EVEN`; this causes the assembler to skip a byte if it is currently at an odd byte. Assembler language instructions, `.WORD` directives, and `.END` statements must all be assembled at word addresses.

```
MEM:      .ASCIZ      /WAITING FOR GODOT/
          .EVEN
          .END        START
```

In this example, if the `.ASCIZ` directive started on a word boundary, the `.EVEN` directive is not actually necessary (do you believe this?). Nevertheless, it is quicker and a good deal more reliable to put the directive in than to count bytes.

EXERCISE: What would result from the sequence

```
MEM:      .PRINT      #MEM
          .ASCIZ      'I'm all right'
          .END        START
```

and what should be done about it?

Local Symbols Complex programs with a great number of subroutines may well tax the programmer's inventiveness in creating labels. PDP-11 assembler language offers the convenience of special "reusable" labels. Consider the following input routine, where we have used the CoMPare a Byte instruction `CMPB`.

```
READ:     MOV         #MEM,R1
1$:       .TTYIN      (R1)
          CMPB        #12,(R1)+
          BNE         1$
DONE:     ...
```

(How does this routine store the input text?) The expression `1$` (\$ is just the dollar sign, not `ESCAPE`) is such a "local label." Local labels must be of the form `1$, 2$, 3$`, and so on. Local labels may be referenced in exactly the same way as "ordinary labels," *as long as no ordinary label intervenes* between the line bearing the local label and the line referencing it. The label `1$` in our illustration is local to the program block between `READ` and `DONE`; outside that block *it will not be recognized at all*. Thus an instruction following `DONE`, preceding `READ`, or both of these could also bear the label `1$` without ambiguity. Any instruction referencing such a label is always interpreted by the assembler as referring to the one in its own local block and is encoded accordingly.

Note that local labels must label a word address, not an odd byte address. This restriction does not apply to ordinary labels.

3.2 STACKS

A *stack*, as defined in Section 3.1, is a collection of items stored in such a way as to make the most recently stored item the most readily accessible. Stacks are extremely important for storing data in a computer. Successive quotients in a printout routine and (as we have seen) the start addresses of successively nested subroutines are two of the many examples of information that is handled in just such a “last in, first out” fashion.

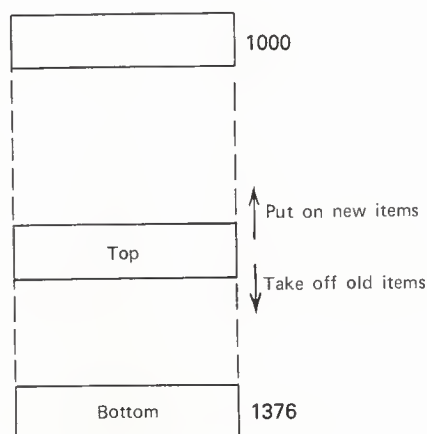
Stacks have acquired a variety of names over the years of their usage. They are commonly called *pushdown lists*. This name arises by analogy with the type of plate holder found in cafeterias. An extra plate is “pushed down” onto the top; in computer terminology, one obtains a plate by “popping” it up from the top. These terms can easily convey the false impression that a whole list of stored items in computer memory is moved when a new item is pushed down or the latest addition is popped up.

It is preferable to say that a new item is *put on*, or the last one *taken off*, the *top* of a stack. At the *bottom* of the stack is the first item put on, which will be the last to be taken off. Even this terminology can be misleading, as we shall see, but less seriously so.

The Stack Pointer A stack on the PDP-11 is just an area of memory set aside by the programmer or by the operating system for storing data that will be wanted on a “last in, first out” basis. Data stored in stack locations look just the same as data stored anywhere else and, indeed, may be accessed in the same way. In addition, however, a block of memory locations that warrants the name of stack will do so because the program maintains a *pointer* to the top of the stack. Putting an item on or taking an item off the stack will be achieved by indirectly addressing the stack pointer.

Suppose that a program uses **R1** as stack pointer and has set aside locations 1000 up to, but not including, 1400 for use as a stack. This simply means that the program avoids referring to these locations except indirectly via **R1**. Initially, since nothing has been calculated, the stack is empty. The program must begin by setting up **R1** so that items are put onto the stack starting from the location designated as its bottom.

It is customary on the PDP-11 to treat the numerically highest stack address as the bottom of the stack. As the illustration shows, this is reasonable enough. In any case, this convention should be followed for reasons that will be stated shortly.



Since stacked items are to be placed in locations of successively numerically lower address, autodecrementing is called for. Thus initially we should **MOV #1400,R1**. Thereafter, the contents of **MEM** can be put onto the stack by

```
MOV      MEM, -(R1)
```

and an item can be taken off the stack and put into **WRD** by

```
MOV      (R1) +, WRD
```

In each case, **R1** is left pointing to the new top of the stack.

It may seem somewhat misleading to talk of taking an item off the stack, when the **MOV** instruction with destination **WRD** has no effect on the location housing the data supposedly removed from the stack. The point is that an item is taken from the stack not by clearing the location housing it, but by moving the stack pointer so that the location is no longer considered to be part of the stack. If an item is now put onto the stack, it will go into this newly freed location.

It is convenient for a program to use a block of words immediately preceding its start address for the stack. Thus, assuming that our program is to be loaded starting from location 1000, we could set up the stack and initialize **R1** by

```
START:    .BLKW      400
          MOV        #1400,R1
```

There is, however, no need for a program to be cognizant of its start address in order to set up such a stack. The purpose of the **MOV** instruction is to move the address of the first word of that instruction itself into **R1**. PDP-11 assembler language recognizes the symbol **.** (a period) as meaning the first word of the instruction in which it is used. Thus we could replace the **MOV** instruction with

```
START:    MOV        #.,R1
```

The symbol **.** can be used in branch instructions, taking advantage of the fact that *the assembler can perform arithmetical calculations*. For example,

```
TST      (R0) +
BEQ      . - 2
```

repeats a loop until a nonzero data item is encountered. *We strongly advise against using this notation*. Any convenience is far outweighed by the danger of miscounting, which is particularly great because of the variable length of PDP-11 instructions. In fact, with local labels of the form **n\$** available, using **.** in this way really offers no countervailing benefits at all.

It is, however, convenient to be able to use this power of the assembler in forms such as **MEM + 2** (the address of the word following **MEM**), **MEM - 2**, and so forth. What problems do you suppose might result from a reference to **MEM + WRD** or **MEM - WRD**?

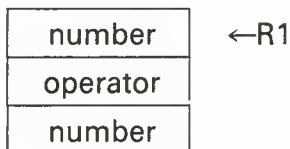
The assembler will also perform multiplication (denoted by *****) and division, discarding any remainder (denoted by **/**). Note, however, that the assembler calculates arithmetical expressions from left to right without observing the usual priority among arithmetical operators. Thus, to the assembler, **1 + 2 * 2** is equal to 6, not 5. Pairs of *angle brackets* **< . . >** may be used to indicate a different order of computation.

Evaluating Arithmetical Expressions Let us consider how to compute arithmetical expressions in the way that the assembler does. Our goal is a program to read such an expression typed in

at the terminal and to print out the value of the expression. Such an expression will be a sequence of positive numbers alternating with arithmetical operators; the expression should begin and end with a number. In addition, there may be pairs of parentheses; a left parenthesis must immediately precede a number, and a right parenthesis must immediately follow a number. Thus $3*(7-4)+1-(6-3)$ is such an expression. Note that we do not use parentheses to indicate multiplication. (Also remember that an expression included in an assembler language program must use angle brackets, not parentheses.)

Our program will use a stack, with R1 as stack pointer. We shall begin by reading the expression, as ASCII text, into a block of locations and pointing R2 to the first word in the block. We shall need some utilitarian subroutines, which you can write for yourself. Subroutine NUM will compute the number whose first digit is at (R2) and leave R2 pointing to the first location following the one housing the final digit of the number; NUM will put the result of its computation onto the top of the stack. Subroutine OP will put the arithmetical operator (in the form of its ASCII code) found at (R2) onto the top of the stack and leave R2 pointing to the next location. Clearly, the program must be sure that R2 points to the first digit of a number when it calls NUM and to an operator when it calls OP.

There must be a subroutine CALC to perform calculations when the following situation is reached on the stack.



CALC must perform the operation required, put the result where the bottom number of the two is, and remove the other two items from the stack. For example, if the operator were known to be +, CALC would perform

```
TST      (R1)+,(R1)+
ADD      -4(R1),(R1)
```

Note the directions on the stack! Of course, we do not know in advance what the operator is going to be, so subroutine CALC must begin by checking the contents of location 2(R1) and then branch to a routine to perform the appropriate calculation.

EXERCISE: Write these utilitarian subroutines.

The heart of the program will be a subroutine EVAL to evaluate an expression. The main program will read the expression from the terminal into the memory block pointed to by R2, call EVAL, call a printout routine, and exit. The printout routine should print out the contents of the bottom stack location.

Let us consider first how to write EVAL to deal with expressions without parentheses. We compute the first number and put it on the stack. If the following symbol is a $\hookleftarrow$, there is nothing to compute, and we return to the main program. Otherwise, we put an operator and the next number onto the stack. We then call CALC to perform the indicated calculation, leaving the result at the bottom of the stack. If a $\hookleftarrow$ now follows, we return to the main program. Otherwise, we put the next operator and number onto the stack and repeat the process.

```
EVAL:      JSR      R5,NUM
            CMP      #15,(R2)
            BNE      1$
            RTS      R5
```

```

1$:      JSR      R5,OP
         JSR      R5,NUM
         JSR      R5,CALC
         CMP      #15,(R2)
         BNE      1$
         RTS      R5

```

- EXERCISES:**
- (i) How much stack space will the program need?
 - (ii) Draw a flowchart for the program.
 - (iii) Write the program, annotating it with suitable comments throughout.

Let us follow through the workings of this subroutine. Suppose we type $1 + 2 * 2$ ↵. **EVAL** starts by calling **NUM**, thus putting 1 onto the stack. Since the next character is not ↵, the program branches to 1\$ in **EVAL** and puts the operator + and the number 2 onto the stack. The stack now looks like this.

2	← R1
+	
1	

Next we call **CALC**, which returns, leaving the stack like this.

3	← R1
---	------

The process of putting operator and number onto the stack is now repeated, giving

2	← R1
*	
3	

which **CALC** reduces to

6	← R1
---	------

The next character is ↵, so **RTS R5** returns us to the main program. The routine to print out the bottom element of the stack now picks up the 6, and the program exits.

EXERCISE: Amend the program so that it can deal with a $-$ sign preceding the first number in the expression or following a left parenthesis (*unary* minus, as opposed to the *binary* minus operator between numbers).

Recursive Subroutines The crucial development in our program is to enable it to manage parentheses. Consider how we compute an expression such as $3 * (7 - 5)$. We begin by going through the same steps as in the **EVAL** subroutine, noting that 3 is the first number and that it is to be multiplied by something. Then we encounter the left parenthesis, and we immediately put aside the $3 *$ calculation and start a new one. The new calculation is to evaluate $7 - 5$; to do this, *we go through the EVAL routine*

again, but now with 7 as the starting number. The **EVAL** routine for this subsidiary computation is terminated when we reach the right parenthesis, having used **CALC** to return the value 2. We can now go back to the original calculation, knowing that $3*2$ is the computation to be performed; the original **EVAL** routine calls **CALC** to do this and returns to the main program.

After putting the first number and operator onto the stack, we may encounter either another number or a left parenthesis. In the latter case, we want to restart the evaluation process; in other words, *we want to call subroutine EVAL*. Thus **EVAL** will provide an example of a subroutine that calls itself—a *recursive* subroutine.

This is a good point at which to refer back to Section 3.1 and check that there is nothing in the workings of the **JSR** – **RTS** mechanism to prevent a subroutine calling itself. You should work through the discussion around Figure 3.2 and satisfy yourself that the correct return addresses are reached even if **SUB1**, **SUB2**, and **SUB3** are replaced by the single subroutine **SUB**. Of course, a recursive subroutine must call itself on some conditional basis; otherwise, it would nest indefinitely and never make its return.

Recursive subroutines do present an enhanced version of a danger common to all subroutine use: care must be taken that the subroutine does not disturb locations that the calling routine is going to use. Consider, for example, what would happen if **EVAL** kept the running total of its computation in **R0** and proceeded to call itself. The way we have written **EVAL**, however, enables us to develop it as we wish while avoiding this danger; **EVAL** uses the stack pointed to by **R1** for its running total and intermediate data so that each recursive call has new locations available for its use.

The ultimate effect of our present **EVAL** subroutine is to put the result of its computation onto the stack before returning. If we take care that this remains the case when we extend the power of **EVAL**, then the net effect of calling **EVAL** appears to the calling routine as similar to that of calling **NUM**: a number appears on the stack. The importance of this observation is that it enables us to keep **EVAL** structurally quite simple. A flowchart for this routine appears in Figure 3.3 on page 74. The conditional branch boxes refer to the next symbol to be processed, as before.

EXERCISE: Apart from realizing that on return from **CALC** a right parenthesis may be the next symbol encountered, we have not properly considered the return conditions for the new version of **EVAL**. Does this limit the range of expressions we can handle? In particular, could our routine, as flowcharted in Figure 3.3, evaluate

- (a) $(1+2)*3$?
- (b) $1+((2*3)+4)$?
- (c) $1+(2*(3+4))$?

How would you amend the structure of **EVAL** to deal with these considerations?

Now we must write **EVAL** to conform to the requirements of the flowchart. Such a routine appears in Figure 3.4. Let us follow it through for the computation of $3*(7-4)-(1-(6-3))$. Figure 3.5 displays the state of the stack at various stages of the computation; we shall refer to these states by the letters beneath them in the figure. These two figures appear overleaf.

EVAL begins by putting 3 and * on the stack, as at *a*. It then encounters a (so, remembering first to point **R2** to the character after the (, it calls **EVAL**. This call of **EVAL** puts 7, -, and 4 successively on the stack, as at *b*; it then calls **CALC**, which computes $7-4$, and leaves the stack as at *c*. Refer back to your **CALC** subroutine and check that this is indeed its function. **EVAL** now encounters a) and so returns. Note how the **CMP** instructions after **3\$** use automatic stepping to ensure that **EVAL** does not return pointing to the) and also that a $\leftarrow$ is not passed over. The return is to the line after **JSR R5,EVAL** in subroutine **EVAL**; remember that the “main” **EVAL** has not yet reached its end. It is clear from both the program and the flowchart that **EVAL** now calls **CALC**, which computes $3*3$ and leaves the stack as at *d* (octal notation).

The “main” **EVAL** now branches back to **1\$** and puts - on the stack, as at *e*; then, encountering

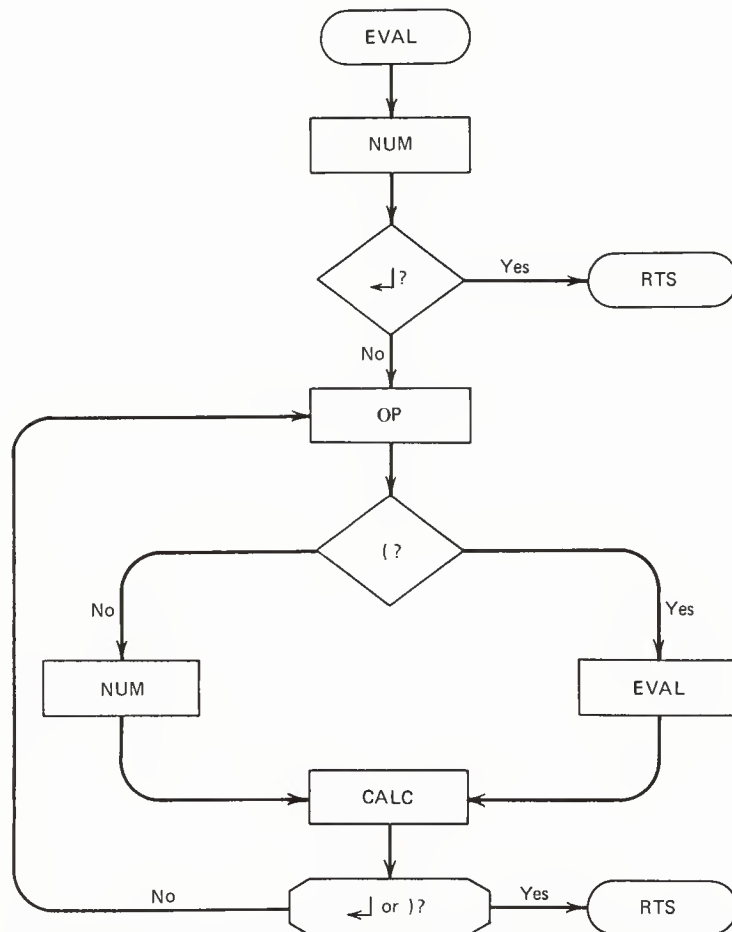


Figure 3.3 A flowchart for evaluation of arithmetical expressions.

a (, it calls **EVAL**. This “subordinate” **EVAL** puts 1 and – on the stack, as at *f*, when, encountering another (, it calls **EVAL**. This “sub-subordinate” **EVAL** puts 6, –, and 3 on the stack, as at *g*; calls **CALC**, which leaves the stack as at *h*; and returns to the line in **EVAL** after **JSR R5,EVAL**. In terms of the stack of subroutine return addresses maintained elsewhere, this return is to subordinate **EVAL**. Subordinate **EVAL** now calls **CALC**, which leaves the stack as at *i*; then, encountering a), it returns to main **EVAL**. Main **EVAL** calls **CALC**, leaving the stack as at *j*, and, encountering the final ↵, returns to the main program and so to the printout routine.

- EXERCISES:**
- (i) Write the complete program.
 - (ii) Amend the program to allow left parentheses to begin the expression, as discussed earlier.
 - (iii) Amend the program to allow the use of unary minus.
 - *(iv) Amend the program so that, unless parentheses demand otherwise, the order of computation will give * and / priority over + and –, as is customary.
 - (v) Incorporate routines to print out error messages if the parentheses in an expression are not properly paired.

The System Stack As the discussion of the **JSR** and **RTS** instructions may have indicated, the PDP-11 system hardware makes use of a stack. As stack pointer the hardware always uses register

```

EVAL:  JSR      R5,NUM
        CMP     #15,(R2)
        BNE     1$
        RTS     R5
1$:     JSR      R5,OP
        CMP     #'',(R2)
        BNE     2$
        TST     (R2)+          ;move past the (
        JSR     R5,EVAL
        BR      3$
2$:     JSR      R5,NUM
3$:     JSR      R5,CALC
        CMP     #'',(R2)+      ;move past the )
        BEQ     4$
        CMP     #15,-(R2)      ;wasn't a ) , so move back
        BNE     1$
4$:     RTS     R5

```

Figure 3.4 A recursive subroutine to evaluate arithmetical expressions.

number 6. This register is called the *hardware stack pointer*, or often just the stack pointer, and is conventionally assigned the name **SP**; you will generally need to declare in your program that

SP=%6

We now know that the instruction

JSR R5,SUB

is equivalent to

MOV R5,-(SP)
 MOV "return PC",R5
 MOV #SUB,PC

while **RTS R5** is equivalent to

MOV R5,PC
 MOV (SP)+,R5

Remember, however, that the multiple tasks of **JSR** and **RTS** are implemented in a single operation by the hardware. These instructions are not “translated” into the sequences given as their equivalents, and these sequences should be regarded as descriptive, not definitive.

						3				
						—				
	4					6	3			
	—					—	—			
	7	3			1	1	1	-2		
×	×	×		—	—	—	—	—		
3	3	3	11	11	11	11	11	11	13	
a	b	c	d	e	f	g	h	i	j	

Figure 3.5 The stack during evaluation of $3 \times (7 - 4) - (1 - (6 - 3))$.

Virtually every operating system will reserve space for the stack and initialize **SP**. The system stack normally starts at location **776**, and the space available to it extends downward to location **400**. User programs are then loaded immediately beyond the stack area, starting at location **1000**.

A program run on a standalone machine without an operating system that sets up the system stack should begin with

```
MOV      #.,SP
```

to initialize the stack. Care must also be taken that room is left, in memory preceding the program, for the stack; this will be discussed at the end of this section.

User programs may reference **SP** in the same way as other registers, *except that, like PC, the contents of SP must be even so as to point to a word address*. In particular, programs may use **SP** as their own stack. This is generally a good idea: with **SP** and **PC** serving their special functions and another register taken up as linkage pointer, a program can ill afford to devote one of the remaining five registers for its stack. The result is that subroutine return addresses and data occupy the same stack. This will often allow greater programming flexibility although, on occasion, it threatens enough confusion to warrant the use of a separate program stack.

EXERCISE: Adapt our **EVAL** routine for a program using **SP** instead of **R1** as stack pointer. Analyze the computation of the expression just considered and prepare a chart of stack states, as we did in Figure 3.5. Do you think that this improves the program?

The astute reader may wonder, if we are concerned with economizing on register usage, why we have to maintain a separate register for subroutine calls. Why not just put the return **PC** onto the stack when calling the subroutine and remove it for the return? This is indeed possible, and is achieved by referencing **PC** as linkage register in the call

```
JSR      PC,SUB
```

and on the return

```
RTS      PC
```

In Section 3.4 we shall discover the benefits of maintaining a separate linkage register.

Coroutines This topic is too complex to be treated here in any detail, yet it is managed too elegantly on the PDP-11 to be bypassed altogether.

We have seen that a **JSR** instruction using **PC** as linkage register moves the effective address calculated for the **JSR** into **PC** and puts the return **PC** onto the system stack. An interesting form of **JSR** instruction results when the address of the routine to which control is to be transferred has itself been stored on the stack. Note that in this case the desired effective address is not (**SP**), which is a location in the stack area, but rather the location pointed to by (**SP**). Thus control is transferred by

```
JSR      PC,@(SP)
```

This puts the return **PC** onto the stack, on top of the address to which control has just been transferred. There is, however, no reason why this latter address should now remain on the stack. It would be better to remove it from the stack before putting **PC** onto it. We can do this by amending the instruction to

```
JSR      PC,@(SP) +
```

This instruction interchanges the contents of PC and the top of the stack. (What does this imply about the order of operations within the JSR instruction?) It follows by symmetry that the routine to which control has been transferred can restore control to the calling routine by issuing exactly the same instruction. After this, the stack will contain the address of the location following that at which the second routine gave up control. Once again, JSR PC,@(SP)+ will return control to this location.

We have a situation in which two routines can transfer control back and forth between them. Such routines are called *coroutines*, indicating a state of cooperation between equals rather than the servitude of a subroutine.

Coroutines are invaluable when a program divides into two major tasks that must be coordinated but would be excessively complicated if coalesced. The routine for the task that begins first will put the address of routine 2 onto the stack. Thereafter, whenever either routine requires the help of the other, it issues a JSR PC,@(SP)+. System programs often perform such complex interwoven tasks and therefore use coroutines. Figures 3.6 shows a simplistic breakdown of the twin processes of reading and interpreting that an assembler goes through when it encodes an instruction.

Location Counter The symbol . represents the *location counter* maintained by the assembler during the process of encoding a program. We have encountered the use of . in an instruction operand to denote the first word of the instruction. Thus a program can initialize the system stack by

MOV #.,SP

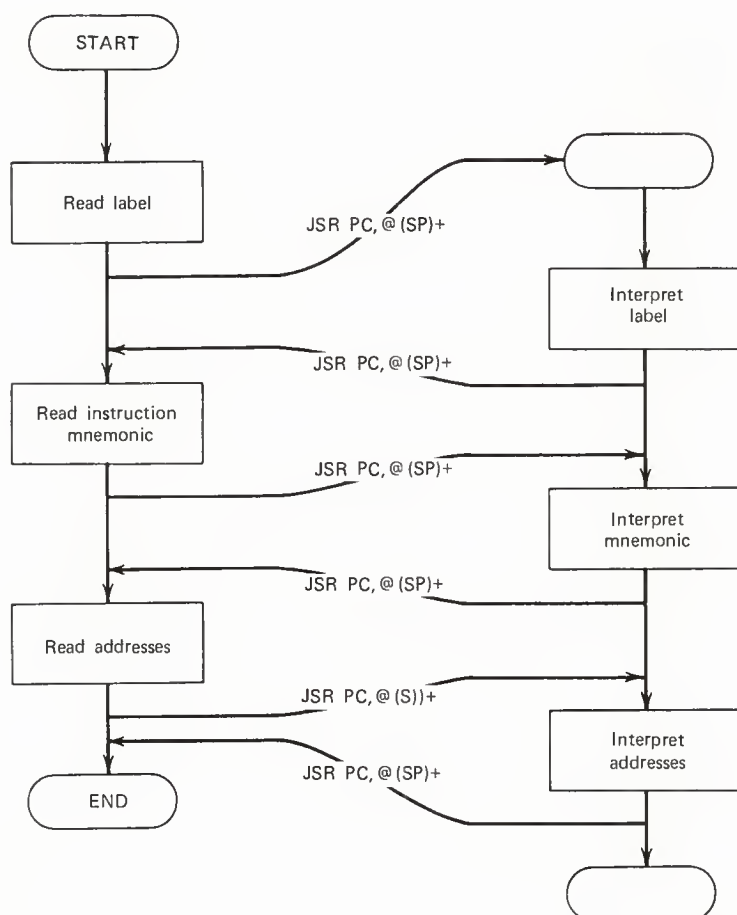


Figure 3.6 Coroutine control transfers in assembly process.

However, this is of no use unless there will be room in memory preceding the program. Most operating systems will guarantee a certain amount of room by loading the program from address 1000 onward. If more room is required than the system automatically provides, the program can specify a higher start address by beginning with a suitable `.BLKW` directive. A `.BLKW` directive instructs the assembler to increment the location counter by twice the number specified (the location counter counts bytes). An alternative to `.BLKW` is the `.BLKB` directive to reserve a BLock of Bytes. Thus

```
.BLKW      40
```

and

```
.BLKB      100
```

are equivalent. Another way of achieving the same result is to give `.` a new value in a direct assignment statement.

```
. = . + 100
```

It is also possible for a program to specify its load address directly by setting the actual value of the location counter in a direct assignment statement. We may, for example, want to have our program loaded starting at location 1400. Generally, however, we cannot just start the program with

```
. = 1400
```

because most assemblers (although not all) regard all code as relocatable unless declared otherwise. The location counter is then also relocatable, equal at the start of assembly to relocatable 0. The ultimate value of relocatable 0 will depend on what is done by the linking loader; it cannot be tied down to 1000 during the assembly process, and any attempt to do so will be regarded by the assembler as an error.

The assembler can, however, be instructed to encode using absolute instead of relocatable addresses by the directive

```
.ASECT
```

After this, `.` is also regarded as absolute and so may be set equal to any desired address.

```
. = 1400
```

We shall see in later sections that a program may need to load certain low-numbered memory locations that control the operations of various hardware functions. Suppose, for example, that location 12 must be set to contain 340, but that otherwise a normal relocatable program is to be written. Instead of loading this location at run time (how?), it can be done during assembly.

```
.ASECT
. = 12
.WORD      340
```

The directive

```
.CSECT
```

then instructs the assembler to regard all ensuing code as relocatable, and we can proceed with the program.

A program may contain several alternating **.ASECT** and **.CSECT** directives, each introducing a section of absolute or relocatable code. The assembler maintains separate location counters for the two kinds of code, so the first line of a new **.CSECT** section will automatically be loaded immediately following the last line of the previous **.CSECT** section.

- EXERCISES:**
- (i) Which method of loading location 12 do you prefer?
 - (ii) What does **MOV .,SP** do?
 - (iii) How are **.** and **PC** related?
 - (iv) The instruction **MOV #.,SP** is eventually loaded starting at location 1400. How is it encoded. How was it encoded at the assembly stage? Do your answers depend on whether it was in an **.ASECT** or **.CSECT** section?

3.3 CONTROL TRANSFER DECISIONS

A structured program, in which a number of subroutines are coordinated to accomplish a complete task, must at various points make decisions as to where, or whether, control is to be transferred. The only information on which to base these control transfer decisions is the contents of registers and memory locations and the state of the condition code bits.

The decisions will be implemented by means of conditional branch instructions, and it is up to the programmer to present the information provided by the program calculations in a suitable form to the proper branch instructions. For example, a program might be required to check whether the number in **MEM** is even or odd and respond accordingly. There is, however, no branch instruction to test this precise condition. We must first extract the necessary information from **MEM** in a form corresponding to the decision that some branch instruction is able to make. We have the **BEQ** and **BNE** instructions, able to distinguish zero from nonzero items by checking the state of the **Z** bit. So if we can set something to be zero when **MEM** is even and nonzero when **MEM** is odd (or vice versa), our problem is solved. An obvious solution is to divide the contents of **MEM** by two and test the remainder.

Logical Instructions The preceding is a very clumsy solution to the problem of determining whether a number is even or odd. Even before learning a better way, it is clear that much of the computational effort has been wasted: we do not care about the quotient, nor about the exact value of the remainder. A technique that ignores most of what it computes is probably not the best solution to a programming problem!

To achieve a better solution, we must free ourselves from the habit of thinking that what we regard as a number is regarded by the computer as a number. It was this that focused our attention on an *arithmetical* approach, dividing the contents of **MEM** by two.

The contents of **MEM** consist of the state of a collection of bits, and the proper question to ask is: When we encode an even number into **MEM**, what does the bit pattern look like? A moment's thought should persuade you that even numbers encode with bit 0 clear, odd numbers with it set. (Check that it works for negative numbers, too.) We have reduced the problem to that of checking bit 0 of **MEM**.

Certain instructions designed for use when the contents of a word are being regarded as a bit pattern (rather than a number) are called *logical* (rather than *arithmetical*) instructions. The reason for this terminology is that, if we interpret a 1 in a bit as representing "true" and a 0 as representing "false," the contents of a word can be regarded as a collection of truth values; these instructions then

perform the classical logical operations. For example, if the letters p and q represent statements, then in logic the statement " p and q "—written symbolically as $p \wedge q$ —is true precisely when both p and q are true.

On the PDP-11, the Bit Test instruction **BIT** forms the *and* function of the contents of the two operands it addresses. Thus

BIT MEM,WRD

forms $(\text{MEM}) \wedge (\text{WRD})$; this has a 1 in a bit precisely where both (MEM) and (WRD) have a 1. Thus, if **R0** has been set to contain 1, **BIT R0,MEM** will be 0 if (MEM) is even and 1 if it is odd.

BIT forms its result in an internal register, inaccessible to the programmer, without affecting the contents of the operands addressed by the instruction. The condition codes are set according to the value of the result. Thus **BIT** is the logical analogue to **CMP**. Since all addressing modes are allowed, we can test for an even number by

	BIT	#1, MEM
	BEQ	EVEN
ODD:	...	

Since each bit can represent a value "true" or "false," the **BIT** instruction actually performs sixteen logical operations simultaneously. The reader who is unfamiliar with formal logic can regard the logical operations merely as instructions that produce certain specified bit patterns. However, at some stage it will be helpful to acquire a basic knowledge of the elements of symbolic logic. This is especially true for programmers using higher-level languages such as FORTRAN and PASCAL, in which logical operators (such as IF) are specifically available.

To specify the result of a logical operation, we need to know what it does for all possible bit configurations in the source and destination. For each bit we must consider the four possibilities: both source and destination clear; both set; source set, destination clear; and source clear, destination set. We can therefore define such an operation by a configuration table. For **BIT** the configuration table is

Source	0	0	1	1
Destination	0	1	0	1
BIT	0	0	0	1

- EXERCISES:**
- (i) What is the state of the Z and N bits after **BIT MEM,WRD** if the contents of **MEM** and **WRD** are respectively:
 - (a) 100, 200
 - (b) 100, 300
 - (c) 177777, -77777
 - (ii) How would you check whether the contents of **MEM** were divisible by four?

Together with *and*, the basic logical operators are *or* and *not*. The *or* operator requires care, because in everyday speech it is not always clear whether "or" means *inclusive or*, denoted by $\vee$:

" $p \vee q$ " is true exactly when p or q or *both* is true.

or whether it means *exclusive or*, denoted by $\vee$:

" $p \vee q$ " is true exactly when p or q but *not both* is true.

The configuration tables should be compared.

	0	0	1	1
	0	1	0	1
$\vee$	0	1	1	1
∇	0	1	1	0

On the PDP-11, *inclusive or* of two operands is performed by the BIt Set instruction **BIS**. Unlike **BIT**, **BIS** leaves its result at the destination address. Thus

BIS **#20, MEM**

sets bit 4 of **MEM**, regardless of its original state.

Exclusive or is performed by the **XOR** instruction,* which has the peculiar syntax

XOR **R, X**

where **R** is a register and **X** is the destination address. The *exclusive or* function of the contents of **R** and **X** is formed and stored at **X**. Thus

MOV **#20, R0**
XOR **R0, MEM**

will clear bit 4 of **MEM** if it was originally set and will set it if it was clear.

The logical operator *not*, denoted by $\neg$ (also $\sim$), is represented by forming the *logical complement* of the contents of the word: all 1s are set to 0, and vice versa. This is achieved by the single-operand instruction **COM**.

Our last logical instruction is BIt Clear:

BIC **MEM, WRD**

This forms in **WRD** the logical function $\{\neg(\text{MEM})\} \wedge (\text{WRD})$; that is, it clears each bit in **WRD** corresponding to a set bit in **MEM**. Thus consider

BIC **#20, MEM**

This will clear bit 4 of **MEM**.

All of these instructions will set or clear the **N** and **Z** bits according to the value of the result.

- EXERCISES:**
- (i) Write a routine to obtain the value of the *and* function of two operands.
 - (ii) Which instruction interchanges the ASCII codes for the **+** and **-** characters?
 - (iii) Given any two characters, is it always possible to find a single logical instruction that interchanges their ASCII codes?
 - (iv) What is the difference between **COM** and **NEG**?
 - *(v) Write a routine that will successively request input of: the mnemonic for a **BIT**, **BIS**, **XOR** or **COM** instruction; the contents of the source; the contents of the destination. The program should then type out the contents of the destination

*Not available on some smaller CPUs.

and the state of the N and Z bits resulting from implementation of the instruction.

Overflow The PDP-11 has been depicted as implementing two essentially different types of instruction: logical, in which a word is simply regarded as containing sixteen bits; and arithmetical, in which a word is regarded as containing a sign bit plus fifteen bits giving the magnitude of a number.

The PDP-11 CPU is, in fact, somewhat more ambivalent than we have represented. Its **ADD** and **SUB** instructions are oblivious of any difference between bit 15 and the other bits. However, so that arithmetic can be correctly performed, there are hardware facilities that the programmer can use to ensure proper setting of the sign bit.

Suppose we try to add $2^{14} + 1$ to itself. This number is encoded with a 0 in bit 15, since it is positive, a 1 in bits 14 and 0, and 0 elsewhere. The addition is performed as follows.

$$\begin{array}{r} 010 \dots 001 \\ + 010 \dots 001 \\ \hline 100 \dots 010 \end{array}$$

This looks just right for binary arithmetic, until we recall that bit 15 is the sign bit! The bottom line represents not the correct result $2^{15} + 2$, but $-2^{15} + 2$. Note that the result is *not* the negative of the correct result. However, if we regard the bottom line as a sixteen-bit positive number, it is correct.

This situation, in which the magnitude part of the result in an arithmetical operation spilled over to affect the sign bit, is called *overflow*. Clearly, it can also occur with subtraction, as in **SUB MEM,WRD** with **MEM** containing $-(2^{14} + 1)$ and **WRD** containing $2^{14} + 1$.

A somewhat different situation also bears the name of overflow. Suppose first that we perform **ADD MEM,MEM** with **MEM** containing -1 . So every bit in **MEM** is set, and we have

$$\begin{array}{r} 111 \dots 111 \\ + 111 \dots 111 \\ \hline 1111 \dots 110 \end{array}$$

Of course, this cannot take place within the confines of a PDP-11 word. The addition of two sixteen-digit binary numbers, each having leading digit 1, has produced a seventeen-digit result. There is, however, no room for this extra digit, the leftmost in the illustration, in a PDP-11 word, so it must be dropped; we shall discuss later what happens to it, but *it has nothing to do with overflow*. When we drop it, however, we get a sixteen-bit number with the rightmost bit clear and all others set; this is the twos complement representation of -2 , so all is well. Note that the addition was performed as if on sixteen-bit numbers, without any special consideration of the sign bit, but the result has nevertheless turned out correct. This is *not* a case of overflow.

Consider, however, an attempt to **ADD MEM,MEM** with **MEM** containing $-2^{15} + 1$; this encodes with bits 15 and 0 set and the others clear.

$$\begin{array}{r} 100 \dots 001 \\ + 100 \dots 001 \\ \hline 1000 \dots 010 \end{array}$$

Now if we drop the leftmost bit, the result is 2, rather than the correct $-2^{16} + 2$. This situation is called overflow, *not* because the leftmost bit had to be dropped, but because the arithmetical operation did not produce a result with the correct sign. The magnitude of the result is also wrong, but this is a consequence of the overflow, not any part of its definition. Very simply, overflow is defined to occur when the sign of the result in an arithmetical operation is incorrect.

- EXERCISES:** (i) Can the second kind of overflow, with the result positive, occur with subtraction?
 (ii) If adding (positive or negative) numbers m and n causes overflow, will adding $-m$ and $-n$ also necessarily do so?

The CPU is equipped to recognize the occurrence of overflow and will respond to it by setting bit 1 of PS, the *V bit*.

Arithmetical instructions such as ADD, SUB, CMP, NEG, DEC, or INC, which are able to result in overflow, will clear the V bit if no overflow occurs. MOV, CLR, and the logical instructions will always clear the V bit.

EXERCISE: If R0 contains 177776, would you expect TST (R0)+ to set the V bit?

There are branch instructions responsive to the state of the V bit: the Branch if V is Set instruction BVS, and the Branch if V is Clear instruction BVC.

Suppose now that we wish to make a program branch conditional on a comparison between the contents of MEM and of WRD. Regarding these contents as numbers that may be positive or negative, we wish to branch to ONWARD if $(MEM) \geq (WRD)$. We can now see that our earlier approach

CMP	MEM,WRD
BPL	ONWARD

will fail if the subtraction performed by CMP results in overflow. Thus, if MEM contains 2^{14} and WRD contains -2^{14} , most certainly we have $(MEM) \geq (WRD)$. However, CMP MEM,WRD will, because of overflow, form a word with 1 in bit 15 and zeros elsewhere. The BPL instruction has no way of ascertaining that we “meant” this to represent $2^{14} - (-2^{14}) = 2^{14} + 2^{14} = 2^{15}$; the function of BPL is limited to checking the N bit. Now the N bit is set whenever the result of a calculation has its high-order bit set (this is true for the logical instructions as well). Thus the CMP instruction has set the N bit, so the BPL instruction will not cause a branch.

So, when CMP produces a result with the N bit set, before declining to branch we should check whether the 1 in bit 15 of the result represents overflow, rather than a number that is “really” negative. In other words, if the N bit and the V bit are *both* set, we should still branch.

CMP	MEM,WRD
BPL	ONWARD
BVS	ONWARD

Note that, since branch instructions have no effect on the condition codes, one CMP serves for both.

Unfortunately, this will still not do what we want, because it does not take into account the second kind of overflow we considered. Suppose that MEM contains -2^{15} while WRD contains $2^{15} - 1$. Since (MEM) is negative and (WRD) is positive, we certainly have $(MEM) < (WRD)$, and we do not want to branch. But the subtraction is performed as follows.

$$\begin{array}{r} 100 \dots 000 \\ -011 \dots 111 \\ \hline 000 \dots 001 \end{array}$$

This “looks” positive, because bit 15 is clear, so the CPU clears the N bit. However, an arithmetical operation has produced a result with the wrong sign: it should have been $-2^{15} - (2^{15} - 1) = -2^{15} - 2^{15} + 1 = -2^{16} + 1$. So the CPU sets the V bit, and our code will result in a branch.

Thus, when the N bit is clear, before deciding to branch we should check whether the 0 in bit 15 of the result represents overflow, rather than a number that is “really” positive. In other words, if the N bit is clear but the V bit is set, we should *not* branch. In conclusion: instead of our original BPL, we must branch if the N and V bits are either both clear or both set, and we must refrain from branching if one of the bits is clear and the other is set. In logical terms, the condition for a branch is $N \nabla V = 0$. This precise condition is implemented by the Branch if Greater than or Equal instruction, BGE.

CMP	MEM,WRD
BGE	ONWARD

- EXERCISES:**
- (i) Encode the preceding condition using only BPL, BMI, BVS, and BVC instructions.
 - *(ii) Write, *using no instructions referring to the V bit*, a subroutine VSET such that a call JSR PC,VSET will set R0 to contain 1 exactly when the instruction immediately preceding the JSR has set the V bit.

The complementary instruction to BGE is the Branch if Less Than instruction, BLT. Thus BLT will branch if exactly one of the N and V bits is set: $N \nabla V = 1$. These are supplemented by the Branch if Greater Than instruction, BGT, which branches when BGE does, unless the Z bit is set: thus it requires $Z \vee (N \nabla V) = 0$ in order to branch; and by its complementary instruction, Branch if Less than or Equal, BLE.

Carry We have observed that the addition of two sixteen-bit numbers, each with leading digit 1, must result in a seventeen-bit number with leading digit 1, and that a PDP-11 word has no room for this extra bit. When a 1 is “lost” over the leftmost edge of a word, it is said to be *carried* out of the word. The CPU indicates its recognition of this state of affairs by how it sets the *C bit*, which is bit 0 of PS.

The logical instructions, as well as MOV, have no effect on the C bit; note that the V bit is actually cleared by these instructions. TST will clear the C bit, just as it does the V bit. INC and DEC have no effect on the C bit (what do they do to the V bit?).

An ADD instruction will set the C bit when the operation loses a 1 carried out of the word, as just described; otherwise it will clear it.

- EXERCISES:**
- (i) How do CLR, COM, and NEG affect the C bit?
 - (ii) Show by examples that the state of the C bit after an ADD instruction does not indicate whether or not the correct result was produced.

A SUB or CMP instruction will set the C bit when it is necessary to carry a 1 *into* the leftmost bit in order to perform the required subtraction; otherwise it will clear it. Thus consider $5 - 7$.

$$\begin{array}{r} 000 \dots 101 \\ - 000 \dots 111 \\ \hline 111 \dots 110 \end{array}$$

In order to reach the result, a 1 had to be carried into the leftmost bit of the binary representation of 5. The twos complement representation of -2 has a 0 in the rightmost bit, then 1s for as many bits as the word contains. Thus, if an extra bit had been available beyond the leftmost bit, it would have been set to 1 in this computation. So the CPU is reasonable to regard a 1 as having been lost over the leftmost edge and therefore set the C bit. Note that the result is arithmetically correct, so the V bit is cleared.

- EXERCISES:**
- (i) Show by examples that the state of the C bit after a **SUB** instruction does not indicate whether or not the correct result was produced.
 - (ii) If subtraction of a number is regarded as addition of its twos complement, how should the condition for **SUB** or **CMP** to set the C bit be expressed?

There are branch instructions responsive to the state of the C bit: the Branch if C is Set instruction **BCS**, and the Branch if C is Clear instruction **BCC**. These instructions have alternative mnemonics: **BCS** is the same as Branch if LOwer, **BLO**; and **BCC** is the same as Branch if HIgher or Same, **BHIS**.

Suppose we want to clear a block of memory from the word pointed to by **R0** up to and including the word pointed to by **R1**. Before studying this section we might have tried

```

LOOP:      CLR      (R0)+
           CMP      R1,R0
           BPL      LOOP

```

Consider, however, what would happen if we started with **R0** pointing to location 60000 and **R1** to location 170000, assuming that there was enough memory installed for these to be legal addresses. The first **CMP** would be performed with **R0** containing 60002, and the operation would produce 107776. This has a 1 in bit 15, so would set the N bit, and the **BPL** instruction would fail to branch. Once more we have a warning against tacitly assuming that the CPU “knows” what kind of data we are dealing with—here memory addresses, rather than signed numbers.

We wish to repeat the loop until we reach the **CMP** instruction with **R0** containing 170002. The instruction will then perform

$$\begin{array}{r}
 170000 \\
 - 170002 \\
 \hline
 177776
 \end{array}$$

requiring a carry into the leftmost bit of the binary representation of 170000 to complete the calculation and so setting the C bit. While the contents of **R0**, regarded as a sixteen-bit positive number, are no greater than 170000 this subtraction will not require a carry into the leftmost bit, and so will clear the C bit. Thus our code should be

```

LOOP:      CLR      (R0)+
           CMP      R1,R0
           BCC      LOOP

```

In this case the **BHIS** mnemonic is more suggestive than **BCC**.

The Branch if LOwer or Same instructio **BLOS** branches if either C or Z (or both) is set: C/V/Z = 1; its complement is Branch if HIgher, **BHI**.

Setting the Condition Codes A program can set or clear the four condition code bits at its discretion. The operation code for this has 00024 in the leftmost eleven bits (how do we get eleven?). Bits 0, 1, 2, and 3 are set respectively according to whether the C, V, Z, or N bits are to be affected. If bit 4 is 0, the selected bits are cleared; if it is 1, they are set. For clearing and setting individual codes, the mnemonics **CLC**, **CLV**, **CLZ**, and **CLN** and **SEC**, **SEV**, **SEZ**, and **SEN** are recognized. All four condition codes are simultaneously cleared by **CCC** and set by **SCC**. Operation code 000240 is the “no-op” **NOP**; it does nothing at all.

The assembler may be instructed to perform the logical operations *and*, syntax **&**, and *inclusive or*, syntax **!**. Thus

CLV!CLC

sets up a word with a 1 where either CLV or CLC (or both) has a 1. So this combination clears *V and C*.

- EXERCISES:**
- (i) What is the coding for CLV? For CLC?
 - (ii) What would CLV&CLC do?
 - (iii) What would SEV!CLC do

Byte Instructions The instructions MOV, CLR, INC, DEC, NEG, BIT, BIC, BIS, COM, TST, and CMP all assemble with bit 15 clear. If a B is appended to any of these codes (MOVB, CLRB, and so forth), it assembles with bit 15 set and becomes a byte manipulation instruction. With a little care we can use these instructions in the same way as the corresponding word instructions.

Negative numbers are formed within a byte in 8-bit twos complement form. Thus, if MEM is a word address with its contents initially zero, the instruction

MOVB # -2, MEM

would result in MEM containing 000376; while, again with the word MEM initially clear,

MOVB # -2, MEM + 1

would leave the low byte clear and set all but the rightmost bit of the high byte, so MEM would contain 177000. (*Note. Not 376000; the half-word point is not at an octal digit boundary!*)

EXERCISE: What are the contents of MEM after MOVB # -1, MEM + 1 if the word was initially clear?

The condition codes are set according to the byte result. In particular, a result formed in a low byte will set the N bit according to the state of the highest bit of that byte, bit 7 in the word.

The CMPB instruction will set the C bit according to whether it proved necessary to carry a 1 into the leftmost bit of the 8-bit computation. Note that ADD and SUB have no byte equivalents.

Normally, a byte instruction will not affect the other byte in the word of which the destination comprises one-half. The sole exception is a MOVB instruction with destination a register. (Recall that a byte instruction referencing a register can only reference the low byte.) This instruction will move the byte data into the low byte of the register, and also set every bit in the high byte of the register to the same value as bit 7. Thus MOVB to a register *extends the sign bit* of the low byte to the whole word. As long as the datum moved is regarded as a signed integer, the register contents will be correct whether referenced as byte or word.

When automatic stepping modes are used with a byte instruction, the CPU generally will modify the register contents by 1 rather than 2. However, there are important exceptions, for which register contents are incremented or decremented by 2 even for byte instructions: when SP or PC is the register, and when a deferred (indirect) addressing mode is used.

EXERCISE: What do you suppose motivated the PDP-11 designers to build these exceptions into the hardware?

Note that in order for instructions such as JSR and RTS to work, the system stack pointer SP must always point to a word address. So a program that uses a byte stack must employ another register as stack pointer.

The SWAp Bytes instruction **SWAB** swaps the contents of the two bytes at the destination word. If **MEM** contains 000376, then after

SWAB MEM

it will contain 177000. **SWAB** clears the V and C bits and sets N and Z according to the *original* high-order byte of the destination word. Thus **SWAB** will set the N bit according to bit 7 of the result.

Shift and Rotate Instructions Quite often it is necessary to access information buried within the bit pattern of a word. Suppose, for example, we want to know the *parity* of a word; that is, we want to know whether an even or odd number of bits are set to 1 in the word. Parity is often used as an error check by reserving a bit in each word of a file as parity bit and setting or clearing it to ensure that the whole word has, say, odd parity. If the file is later accidentally damaged, it is likely (how likely?) that some word will suffer a parity change, which a later parity check will reveal.

Assume that we wish to check the parity of the contents of **R1**, producing a 1 in **R0** if the parity is odd and a 0 if it is even. We begin by clearing **R0**; then we check each bit of **R1** in turn and, whenever we find a 1, we change the contents of **R0**: to 0 if they were 1, and vice versa. For this action on **R0**, we first set **R2** to contain 1, then when we find a 1 in **R1**

XOR R2,R0

We must still check each bit of **R1**. There are four instructions suitable for sequential bit testing. They are all intimately involved with the C bit.

The Arithmetic Shift Left instruction **ASL** operates on its word destination as follows: the state of bit 15 goes into the C bit; the state of bit 14 goes into bit 15; the state of bit 13 goes into bit 14; and so on, with bit 0 going into bit 1. Bit 0 is cleared. We can illustrate this diagrammatically.

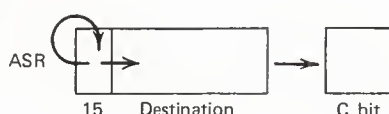


ASL effectively performs multiplication of the destination contents by two. The N and Z bits are set or cleared according to the value of the result. The V bit is set if exactly one of the N and C bits is set; otherwise, it is cleared. Thus $V = N \oplus C$.

- EXERCISES:**
- (i) Does **ASL** perform correct multiplication by two on negative integers in twos complement form?
 - (ii) Does the state of the V bit correspond to the arithmetical correctness of the result?

The byte version of **ASL** is **ASLB**; it shifts the high bit of the byte into the C bit and clears the low bit of the byte. It sets the other condition codes in the same way as **ASL**, but according to the byte result.

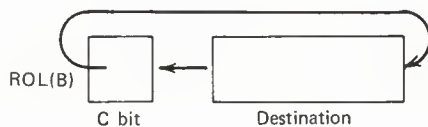
The Arithmetic Shift Right instruction **ASR** loads the C bit from bit 0 of the destination and shifts every other bit value one position to the right. Bit 15 has its value put into bit 14 and is itself not affected.



The byte equivalent, **ASRB**, works analogously. It preserves the state of the high bit of its byte. Both instructions set the condition codes in the same way as the left shift equivalents.

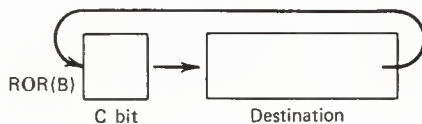
- EXERCISES:**
- (i) What arithmetical operation is **ASR** designed to perform?
 - (ii) Does it perform correctly for all integer values?
 - (iii) Does the state of the V bit indicate the correctness of the result?
 - (iv) Are you now able to complete the parity check routine?

The **ROtate Left (a Byte)** instruction **ROL (ROLB)** performs on its destination the following operation.



Each bit state is shifted one place to the left, with the state of the highest bit going into the C bit; the C bit goes into the lowest bit.

Analogously, the **ROtate Right (a Byte)** instruction **ROR (RORB)** performs the following operation.



Each bit state is shifted one place to the right, with the state of the lowest bit going into the C bit; the C bit goes into the highest bit.

These instructions set the N, Z, and V bits exactly as arithmetic shift instructions do.

EXERCISE: Can you fathom why the rotate instructions were designed to set $V = N \vee C$?

To illustrate the use of rotate instructions, suppose that we know that a double-operand instruction is to be found beginning at location **X**, and that we want to set **R0** to contain the number of the mode in which the source operand at **X** is addressed without altering **X**. Section 2.4 informs us that the source operand mode occupies bits 9 to 11 of the word. Thus one solution is **MOV X,R0** followed by **ROR R0** nine times, so that bits 9 to 11 of **X** occupy bits 0 to 2 of **R0**. Now **BIC #177770,R0** clears all the other bits of **R0**. However, a better solution is

```

MOV B    X+1,R0
ROR      R0
BIC      #177770,R0

```

A fine discussion of the general problem of moving values from part of one word to part of another on the PDP-11 is to be found in Wulf et al., *The Design of an Optimizing Compiler* (American Elsevier, 1975). As the authors point out, no general approach is known, and considerable ingenuity is required to find the most efficient solution in a given case. Study the following method of moving the contents of the rightmost nine bits of location **Y** into the leftmost nine bits of location **X** without affecting **Y** or the rest of **X**.

MOV	Y,R0
ROLB	X
ROR	R0
RORB	X
MOVB	R0,X+1

This code, which occupies nine words in all, is due to the present author. You might like to see if you can improve on it.

- EXERCISES:**
- (i) Write code to set (a) the C bit, (b) the N bit, if the double-operand instruction at *X* uses a deferred mode to address the source; clear the bit otherwise. Leave the contents of *X* unaltered.
 - (ii) Write code to set the contents of *R0* equal to the number of words taken up by the double-operand instruction beginning at location *X*.

Encoding Branch Instructions All branch instructions assemble into one word of code. The high byte contains the operation code for the particular conditional branch, while the low byte determines the destination of the branch. The destination is expressed in the form of an *offset* from the current contents of *PC*. Thus, suppose at location 2000 we had the instruction **BNE LABEL**, where **LABEL** is a label attached to location 2010. When the branch instruction is performed, *PC* contains the address of the following word, 2002. The offset between the instruction and its destination is $2010 - 2002 = 6$ bytes (octal!), or 3 words. The assembler encodes the *word* offset into the low byte of the instruction. (Of course, the CPU takes this into account when it performs the instruction.) The operation code for **BNE** has bit 9 set and the rest of the high byte clear, so here **BNE LABEL** encodes as 001003.

The low byte of a branch instruction expresses the offset as a positive or negative number, with the latter in 8-bit two's complement form. Suppose we wish to place at location 2010 the instruction **BNE LOOP**, where **LOOP** is location 2000. When the branch instruction is performed, *PC* contains the address of the following word, 2012. Thus the offset between the instruction and its destination is $2000 - 2012 = -12$ bytes, or -5 (octal!) words. Note that a backward offset is regarded as negative. -5 encodes in the low byte as 373, so **BNE LOOP** encodes as 001373. Seeing such a code in a program listing, we would recognize a branch *backward*, by *five words*, starting the count *from the word following the branch instruction*.

Observe that a low byte of 177 gives the maximum forward branch of 200 (D 128) words from the instruction itself; a low byte of 200 represents -200 and gives the maximum backward branch of 177 (D 127) words from the instruction itself. A single branch instruction cannot transfer control beyond this range.

- EXERCISES:**
- (i) The **BEQ** instruction has operation code 0014 in the leftmost eight bits of the word. How would **BEQ LABEL** encode at location 2006, if **LABEL** were (a) location 2000, (b) location 1712, and (c) location 2076?
 - (ii) Are branch instructions position independent?

There remains one last branch instruction, Subtract One and Branch if nonzero, **SOB**.* This has operation code 077 in bits 9 to 15, a register reference in bits 6 to 8, and its offset in bits 0 to 5. This offset is calculated as a 6-bit positive number, but it is interpreted as a *backward* offset. Thus **SOB** cannot branch forward. The instruction will subtract 1 from the register referenced and will

*Not available on some smaller CPUs.

branch if the register contents, after the subtraction, are nonzero. **SOB** is a useful instruction for loops; we can clear memory from **MEM** up to and including **WRD** by

```

                MOV      #MEM,R0
                MOV      #(WRD - MEM)/2 + 1,R1
LOOP:          CLR      (R0)+
                SOB      R1,LOOP

```

- EXERCISES:**
- (i) What is the maximum range of a **SOB** instruction?
 - (ii) How is the **SOB** instruction in the previous example encoded?
 - (iii) Write a routine to clear:
 - (a) 1000 locations up to and including **MEM**.
 - (b) Locations **MEM - 1000** up to and including **MEM**.

3.4 MODULAR PROGRAMMING

This section develops further the theme of Sections 3.1 and 3.2, breaking a programming task down into more tractable sections, or *modules*. Our particular concern will be with the linkages between modules.

When we wrote our program of Section 3.2 to evaluate arithmetical expressions, we decided that the expressions should be read into a block pointed to by **R2** and that the calculations should be performed on the stack pointed to by **R1**. Thus, for example, subroutine **NUM** was designed to collect its data from locations starting at **(R2)** and deliver its results to **-(R1)**. This effectively makes **NUM** a general routine to read a number, not merely one confined specifically to the needs of that program. We can use **NUM** to pick up digits from anywhere in memory and deposit the resulting number anywhere in memory. All we need to do is set up the pointers **R1** and **R2** before calling **NUM**.

Passing Parameters The items of information that must be made available to a subroutine so that it can perform its function are called the *parameters* of the subroutines. The calling procedure must *pass* the parameters to the subroutine. The return address is clearly, by our definition, a parameter, and we have already discussed how this is passed by the **JSR** instruction. There are several methods for passing the parameters that tell the subroutine where it should get its data and deliver its results. None of these methods presents any difficulty to the assembler language programmer, who can see exactly what is going on. It is much easier for programmers in higher-level languages, where the mechanical details are obscured, to become confused about what the various parameter passing techniques actually do.

Suppose that we are writing a subroutine **SQRT** to calculate the square root of a number; consider how we might pass parameters to it. The subroutine might be written to form the square root of the number in the location pointed to by **R1** and put the result in the location pointed to by **R2**. This is similar to the **NUM** situation. To form in **WRD** the square root of the number in **MEM**,

```

                MOV      #MEM,R1
                MOV      #WRD,R2
                JSR      PC,SQRT

```

To leave the original number undisturbed in **MEM**, **SQRT** should begin with something like

```

                MOV      (R1),R1

```

If the square root is calculated in R1, the subroutine can conclude with

```
MOV    R1,(R2)
RTS    PC
```

Clearly, it is not hard to adapt this so that the square root of the number found in MEM is put into MEM, if this is desired.

Another way of passing parameters is to list them in the program after the JSR instruction, using a linkage register other than PC.

```
JSR    R5,SQRT
.WORD  MEM,WRD
```

The subroutine then picks up its data into R1 by

```
MOV    @(R5)+,R1
```

and deposits the result from R1 by

```
MOV    R1,@(R5)+
```

Autoincrementing R5 ensures that the parameters are passed in sequence; it also allows the return to be made by RTS R5 to the instruction following the last parameter. Obviously, PC cannot be the linkage register for this method of passing parameters. However, if parameters are not being passed in this way, there is rarely any reason to expend another register on subroutine linkage.

We could even follow the JSR with the actual number whose square root is to be put into WRD.

```
JSR    R5,SQRT
.WORD  7                      ;to form root of 7
.WORD  WRD                    ;and put it in WRD
```

as long as the subroutine picks up the 7 by

```
MOV    (R5)+,R1
```

It is also feasible to put the parameters onto the system stack before calling the subroutine. The obvious precautions must be taken to ensure that SP points to the return address and not to a parameter when the return is made. Another difficulty is to avoid leaving the parameters on the stack, which would soon fill up the stack area. It may be necessary to have the subroutine “clean up” the stack by resetting SP according to the number of parameters passed. Some careful counting is needed to avoid falling prey to either of these dangers.

- EXERCISES:**
- (i) What considerations would lead you to prefer one of these parameter passing techniques to the others? In particular, which is best for:
 - (a) Nested subroutines?
 - (b) Recursive subroutines?
 - (c) Subroutines referencing a large number of parameters?
 - (d) Subroutines referencing a variable number of parameters?
 - (ii) Polish to perfection your subroutines to:
 - (a) Multiply two numbers together.

- (b) Divide one number by another.
- (c) Read a decimal number from the terminal.
- (d) Print out a decimal number at the terminal.
- (iii) How should coroutines pass parameters? Is there any need to clean up the stack after them?

Parameters in Higher-Level Languages Assembler language programmers are concerned with finding a consistent and efficient method for linking subroutines and must pay close attention to technical details. Higher-level language programmers, on the other hand, are spared any concern with the technicalities of subroutine linkage, but are in danger of falling into logical error if they do not understand the linkage system employed by the language they are using.

To higher-level language programmers, it matters little which of the techniques just discussed for passing parameters is employed. Their concern is not with how the subroutine gets its information but with what information it gets and how it deals with it.

When programming in a higher-level language, it is easy to think wholly in arithmetical terms, of variables and constants. This is encouraged by languages with syntax constructions such as

$$X = 3$$

conveying the impression that some intangible entity X has fleetingly settled on the value 3. Languages with syntax such as

$$X \leftarrow 3$$

or

$$X := 3$$

convey the underlying reality more vividly; rather than “set X equal to 3,” one reads such a line as “put 3 into X .” When the programmer declares X to be a variable, the language compiler sets aside a memory location to which it assigns the name X . References to X in the higher-level language are then translated by the compiler into machine language instructions affecting the *contents* of X . Thus

$$Y = X + 1$$

might become

```
MOV    X,Y
INC    X
```

while the seemingly paradoxical $X = X + 1$ is merely the higher-level language version of `INC X`.

The obvious way for a compiler on a PDP-11 to encode $X = 3$ is `MOV #3,X`, and this is indeed the best way. Consider also, however, the possibility of

```
MOV    THREE,X
...
...
THREE: .WORD    3
```

This takes up four words instead of three and is clearly inferior. We mention it only because it represents a technique that, on other computers, offers advantages relevant to this discussion. Not all

computers offer immediate mode addressing. On those that do not, the assembler will respond to a form equivalent to our `MOV #3,X` by reserving a word in a special memory block to hold the constant 3 (the assembler will create a *literal*) and simply translate into the second of the preceding forms. Although considerations of reserving storage for constants are irrelevant on the PDP-11, the point is of interest here because, in many texts, the assumption is made that a higher-level language instruction such as

$$Y = X + 3$$

will always be compiled in a fashion such as

```
MOV    X,Y
ADD    THREE,Y
```

where `THREE` is a location, containing 3, that is accessed every time the program specifies the constant 3. This is indeed the case with good compilers on many computers, since, as explained, it may be more efficient to compile a program in this way. Consider now what would happen if the higher-level language program contained

$$3 = 3 + 3$$

assuming that the compiler did not reject this as an error. The compiler would generate the code

```
MOV    THREE,THREE
ADD    THREE,THREE
```

although a good compiler would include an optimizing stage in which the superfluity of the first line would be recognized. When the program is executed, any subsequent reference to 3 would be treated as if it were 6. In some languages it is easier to destroy a constant in this way than our transparent illustration might evidence.

On a PDP-11, however, the instruction `3=3+3` would, if permitted in the language, be translated into `MOV #3,#3`, and it is harmless. What does this actually do?

Suppose now that in a higher-level language we have a square root function `SQRT` and that we want to be able to instruct

$$Y = \text{SQRT}(X)$$

leaving the value of `X` unchanged. The last requirement means that the subroutine cannot work directly on the contents of location `X`; rather, it will *copy* those contents into its own work space with, say,

```
MOV    X,R0
```

and not refer to `X` thereafter. The parameter `X` is said to be *called by value*.

Note that `Y` is *not* a parameter of the subroutine. The `SQRT` subroutine will be written to leave its result in the location that the compiler uses when the `=` operator is followed by a function name. Thus we would write `SQRT` to leave the square root of the contents of `X` in `R0` if we knew that "`Y =`" preceding a function name is translated as `MOV R0,Y`.

The square root function could be rewritten to be called as a subroutine or procedure. In this case it would be called in a fashion such as

```
CALL SQRT(X,Y)
```

to calculate the square root of the contents of X and put it in Y . Y is now a parameter of the subroutine and is said to be *called by result*. Note that a parameter may be called by both value and result.

The classic example of a subroutine whose parameters should not be passed by value is a procedure to interchange the values of two variables

CALL EXCH(X,Y)

A routine that copies the contents of X and Y into two locations in its own work area, interchanges the contents of these latter two locations, then returns, would not be of much use to us. Instead, the routine must work *directly* with the locations specified in the subroutine call. We say that X and Y must be *called by reference*.

It is possible to produce pathological examples of subroutines whose parameters may be called either by value-result or by reference, with unexpected results in one case or the other. Let us encode subroutine DIFSUM(X,Y), whose higher-level language code is

```
X = X - Y
Y = 2 * Y + X
```

and whose desired effect is to put the difference between the values of X and Y in X and their sum in Y . Note that the second line of code takes into account the new value created in X by the first line. We could call X and Y by value-result:

```
MOV    X,R0           ;call by
MOV    Y,R1           ;value
SUB    R1,R0
ADD    R1,R1
ADD    R0,R1
MOV    R0,X           ;call by
MOV    R1,Y           ;result
RTS    PC
```

or by reference

```
MOV    #X,R0          ;call by
MOV    #Y,R1          ;reference
SUB    (R1),(R0)
ADD    (R1),(R1)
ADD    (R0),(R1)
RTS    PC
```

Now follow through the effect of a call DIFSUM(X,X) in the two cases.

- EXERCISES:**
- (i) If a subroutine has just one parameter, can it make any difference whether it is called
 - (a) by value-result, or (b) by reference?
 - (ii) Study the parameter passing methods available in your favorite higher-level language.

Traps The basic notion of a trap is an automatic intervention by the CPU when certain potentially disastrous situations occur in the course of running a program. On the PDP-11 the concept

is extended to provide a further facility, of interest mainly to standalone system users, for linking program modules.

Try issuing a nonexistent instruction in a program; the "instruction" **.WORD 7** will serve perfectly, since this code is not used for an instruction. The program will stop and, if there is a monitor, it will print out a message that the program has *trapped to location 10*.

When an unrecognizable instruction is encountered, the *hardware* automatically performs the equivalent of

```
MOV    PS, -(SP)
MOV    PC, -(SP)
```

putting the processor status word **PS** and **PC** onto the system stack, and obtains a new **PS** and **PC** by the equivalent of

```
MOV    10, PC
and
MOV    12, PS
```

The choice of locations **10** and **12** is built into the hardware and cannot be changed by the programmer. The nonexistent instruction condition is said to trap to location **10**, where it finds a *two-word trap vector*.

When an operating system is loaded, it will set up location **10** to point to a routine for dealing with this condition. Such a routine is called a *trap handler*, or *trap servicing routine*; in this case it will merely print out a suitable message and return to monitor command level.

Now, as we have observed, on some models of PDP-11 computer **MUL** and **DIV** are nonexistent instructions. A program using these instructions will trap to location **10** on a machine that lacks them. Note that in order to be able to run such a program at all, the *assembler* must recognize **MUL** and **DIV**; otherwise, it cannot encode them. Alternatively, **.WORD** directives can be used to insert the code directly into the program.

Thus programmers wanting to develop a program for possible use on different PDP-11s have a problem. They do not want to rewrite the program when it is transferred to another machine. Indeed, it is wasteful of machine resources even to compile it again on such an occasion; the program should be kept, on tape, portable disk, or the like, in its memory image form, ready for immediate loading and starting. One solution is to dispense entirely with **MUL** and **DIV**. But these instructions are much faster than software routines performing similar functions, so this solution is also potentially wasteful.

A sophisticated way of dealing with this problem is to use **MUL** and **DIV** freely in the program, having set up location **10** to guide operations to the program's own multiplication and division routines. Thus **MUL** and **DIV** will be used when they are available; when they are not, the trap through location **10** will pass control to the user's routines.

On a standalone system or as a privileged user on a timesharing system, location **10** is accessible in the same way as any other memory location. So location **10** can be set up to hold the address of the trap servicing routine **SERV** by **MOV #SERV, 10**. (How else might this be done?) Note that the program should initialize the system stack if there is no monitor to do it (why?).

A Trap Servicing Routine Routine **SERV** must pick up, via the return **PC**, the instruction that caused the trap. It must then check whether **MUL** or **DIV** was issued. **MUL** is encoded as **070** in the leftmost seven bits, followed by three bits for destination register and six bits for source operand. **DIV** is similarly encoded, except that its operation code is **071**. **SERV** must also be prepared to issue a helpful error message if something else caused the trap. Operations must then be directed to the multiplication or division routine, each of which must, of course, leave things exactly as its hardware counterpart would have done.

The division and multiplication routines must pick up the operands for the calculation. For purposes of illustration, suppose that the program contained the instruction

```
X:      MUL      Y,R
```

where *R* is one of the registers, and *Y* is an expression (possibly involving indexing and indirect addressing) denoting an address. The location *X* of the instruction will, of course, have to be picked up by the trap servicing routine via the stack but, for clarity of exposition, we shall assume that we know where it is.

Register *R* might be any of the first six registers; we cannot write instructions using an unknown register (is this *really* impossible?), so we must start by carrying out the equivalent of, say, *MOV R,R0*. We can do this by putting the bits representing *R* in the *MUL* instruction into the source bits of a *MOV* instruction *before* carrying out the latter. Fortunately, this just means transferring bits 6 to 8 from the *MUL* instruction to the *MOV* instruction.

```

                MOV      X,R0
                BIC      #177077,R0
                ADD      R0,L1
L1:            MOV      R0,R0
```

(Would this work if *R* happened to be *R0*?) What impact does this use of self-modifying code have on the statement in Section 3.3 that a compiler should optimize by eliminating *MOV* instructions in which source and destination are the same location?

Picking up the other operand is not quite so simple. As a first attempt, we can put the destination bits (0 to 5) from *X* into the source bits (6 to 11) of a “*MOV* into *R1*” instruction.

```

                MOV      X,R1
                BIC      #177700,R1
                SWAB     R1
                ASR      R1
                ASR      R1
                ADD      R1,L2
L2:            MOV      R0,R1
```

Note that, as before, the original source operand at *L2* is chosen simply to leave the source field clear, ready for the immediately preceding *ADD* instruction to modify it.

If *MUL* was a one-word instruction then, as long as the source operand register was not *R1*, this technique will work perfectly. (How would you adapt it to cover the case when the source operand is *R1*?) Suppose, however, that the source addressing mode led to the *MUL* instruction assembling as two words. This same mode will now appear in the *MOV* instruction at *L2*, which will therefore look to the word *following L2* in its source effective-address calculation. We might deal with this by bringing the second word of the *MUL* instruction over, replacing the last line of the routine with

```

                MOV      X+2,L2+2
L2:            MOV      R0,R1
                HALT
```

Of course, the *HALT* is just a “filler”; the instruction at *L2+6* (where?) ensures that it is not executed.

EXERCISE: * The trouble with this last attempt is that if *MUL* was a one-word instruction, whatever instruction was at *X+2* will be carried over to *L2+2*. Amend the routine so that it

performs `MOV X+2,L2+2` only when `MUL` is a two-word instruction. (You might like to replace `HALT` by `NOP`.)

Even now our routine cannot cope if the `MUL` instruction uses relative (mode 6 with `PC`) or relative-deferred (mode 7 with `PC`) addressing, because then `X+2` will contain the address sought, or a pointer to it, *relative to the instruction at X*.

EXERCISE: * Amend your routine accordingly. This exercise is an excellent review of these addressing modes.

The trap servicing routine should end with the `ReTurn from Interrupt` instruction

`RTI`

which performs in one hardware operation the equivalent of

`MOV (SP)+,PC`

followed by

`MOV (SP)+,PS`

Warning. If a user trap servicing routine is to be used with a monitor present, the contents of its trap vector should be saved at the start of the program and restored before exiting to preserve the path to the monitor's trap servicing routine.

EXERCISE: * Complete the trap servicing routine to "simulate" the hardware multiplication and division instructions.

Absolute Addressing It is possible to avoid relative addressing altogether, by using mode 3 addressing with `PC`. This is called *absolute addressing* and is conveyed to the assembler by preceding the address to be so referenced with the two symbols `@#`. Thus

`CLR @#10`

has the same effect as `CLR 10`, but it assembles as if it were

`CLR @(PC)+
.WORD 10`

(How does `CLR 10` assemble?) When the effective address is calculated, `PC` is pointing to the word containing 10; thus the location defined by `@(PC)` is location number 10, which the instruction clears. Autoincrementing now points `PC` to the next instruction in the program.

Absolute addressing can be used with relocatable addresses. Thus, if `MEM` is location relocatable 140, `CLR @#MEM` will assemble as

`005037
000140'`

(How does `CLR MEM` assemble?)

A program that references locations fixed in memory (such as location 10 here) using relative addressing will not work if its load image is moved elsewhere in memory, but must be linked again

for the new load address. Alternatively, we can use absolute instead of relative addressing, taking advantage of the fact that forms such as `CLR @#10` are *position independent*.

EXERCISE: Is the form `CLR @#MEM` position independent?

If a program has already been written using relative addressing, a nice facility of the assembler enables it to be converted to use absolute addressing without the trouble of rewriting. The directive

```
.ENABL    AMA
```

causes the assembler to treat all relative modes as if they were absolute. Thus conversion is achieved by inserting this line at the beginning of the program and recompiling. It can be helpful to do this while debugging a program, because absolute addressing modes are easier to decipher in a program listing. The `.ENABL AMA` directive will be effective for all program instructions following it or until its effect is turned off by the `.DSABL AMA` directive.

Program-Generated Traps Finding a nonexistent instruction is one of several untoward circumstances under which the CPU will trap to a vector in low-numbered memory locations. In each case, the CPU will put `PS` and `PC` onto the stack and obtain the new `PC` and `PS` from the two-word trap vector.

These are all circumstances under which a trap will occur as an automatic hardware function, regardless of the programmer's wishes. It is also possible, using special instructions, for a program to cause a trap on demand. PDP-11 systems programs make heavy use of the EMulator Trap assembler language instruction `EMT`. `EMT` uses a trap vector at address 30; this means that, after putting `PS` and `PC` onto the stack, the `EMT` instruction causes the new `PC` and `PS` contents to be taken from locations 30 and 32. Most monitor calls include an `EMT` instruction, thereby trapping to a monitor-controlled routine that, at least on timesharing systems, the ordinary user would not be permitted to carry out directly.

The operation code for `EMT` is 104000; this has 0 in the low byte but, in fact, the assembler treats any instruction with the same high byte contents as 104000 as an `EMT` instruction. This has the important consequence that the programmer can use the low byte to convey information to the trap handler. The code to be placed in the low byte is specified as an operand in the `EMT` instruction. For example,

```
EMT      340
```

assembles as 104340, which is actually the RT-11 system macro call `.TTINR`. This monitor call will input a character from the terminal to `R0` and clear the `C` bit as an indication that it has done so; if no character has been typed, this call will set the `C` bit but will not wait. To wait for a character, we need the familiar `.TTYIN` monitor call, which expands into

```
EMT      340
BCS      -2
```

Similarly, we have the `.TTOUTR` monitor call, which is `EMT 341`, and the familiar `.TTYOUT`, whose expansion is

```
EMT      341
BCS      -2
```

We have not included the code used to move the character between **R0** and any other location that might be specified in the call; this will be discussed in Section 3.5.

The operating system will have loaded location **30** with the address of its **EMT** trap handler. The trap handler will begin by saving the contents of **R0** to **R5** on the stack; the return **PC** is then at **14(SP)**. This is brought into **R0**.

```
MOV      14(SP),R0
```

The **EMT** instruction itself is now at $-2(R0)$. The trap handler puts the **EMT** instruction onto the stack.

```
MOV      -(R0),-(SP)
```

A further consideration that enters at this point will be discussed in Section 4.5. Next, the code in the low byte of the **EMT** instruction is put into **R0**.

```
MOVB     (SP)+,R0
```

This removes the now unwanted **EMT** from the stack. Recall that **SP** is incremented by 2, even for byte instructions; the precise effect of these last two instructions could not be obtained with any single instruction.

One way that could now be used to transfer control to the appropriate **EMT** routine is to refer to a *dispatch table*. At location **TABLE** would be a block of 400 words, with each word location $TABLE + \langle 2 * n \rangle$, for $0 \leq n \leq 377$, containing the address of the **EMT** n routine. Thus, after doubling the contents of **R0** (why?), we would

```
MOV      TABLE(R0),PC
```

Control can also be transferred with the **JuMP** instruction **JMP**. The operation code for **JMP** has 0001 in the leftmost four octal digits, leaving six bits for the destination of the jump. A full effective address calculation is carried out. Thus, instead of the **MOV** instruction, we could have

```
JMP      @TABLE(R0)
```

Note that with **JMP**, just as with **JSR**, **PC** is loaded with the destination *address*, *not* with the contents of that address. Because of this, a **JMP** requires one level more of deferral in the addressing mode to achieve the same result as a **MOV** with destination **PC**. Thus **JMP (R0)** has the same effect as **MOV R0,PC**, **JMP @(R0)** has the same effect as **MOV (R0),PC**, and there is no **JMP** instruction to perform the equivalent of **MOV @(R0),PC**. Note that the form **JMP R0** is an "illegal" instruction, since the PDP-11 does not allow control to be transferred to a register; the result is a trap through location 4 (the handbooks for some of the CPUs claim that address 10 is used; we suggest that you check the veracity of this on your own system). **JMP** does not affect the condition codes.

Note also that an instruction of the form **JMP @(R0)+** will load **PC** before incrementing **R0**; the same is true of **JSR**. Is this consistent with our account of **JSR PC,@(SP)+**?

Only programs run without an operating system should write their own **EMT** trap handlers, since DEC software relies on this instruction. User programs may, instead, use the instruction **TRAP**, which is identical in operation to **EMT** except for the address of the trap vector. **TRAP** causes **PC** to be loaded from location 34 and **PS** from location 36. Its operation code is 104400, again with the low byte available for passing information to the trap handler.

Patching We have seen how a program can use **TRAP** instead of **JSR** to call its subroutines. The program must maintain a dispatch table of subroutine addresses, and the subroutines must return with **RTI**, not **RTS**.

Using **TRAP** instead of **JSR** offers the advantage of a one- rather than two-word instruction. This seemingly trivial benefit turns out to be quite useful. It is not uncommon to want to develop a program that has been preserved only in the form of relocatable binary code (**.OBJ**) or memory image file (**.SAV**). There are systems programs designed to ease the task of changing word contents and adding blocks of new code in such files. The new blocks are called *patch areas*, and the technique is called *patching*. In general, however, these systems programs do not offer the facility of moving instructions up to make room for new code to be inserted. Consequently, the new code must be written at the end of the file and an existing program instruction replaced with an instruction to transfer control to the new code. It will obviously be much easier to find a way of slipping in a one-word **TRAP** than a two-word **JSR** (of course, a branch would do if its reach were sufficient). This approach is only possible if there is room in the dispatch table, so it is a good idea to leave plenty of space in this for potential development when the program is first written.

EXERCISE: Investigate the patching programs in your system.

Debugging Aids The BreakPoint Trap instruction **BPT**, operation code **000003**, causes a trap with vector at location **14**. No operand can be included with **BPT**, so no further information can be transmitted by the instruction to the trap handler.

BPT is used by debugging aids such as **ODT**. If you request **ODT** to put a breakpoint at location **X** in your program, it will save the address and contents of **X**, and then

```
MOV      #3,X
```

replacing the instruction at **X** with a **BPT** instruction. When program execution reaches **X**, the trap will occur and give control to **ODT**'s trap handler. At this point, the programmer can issue commands to **ODT**; if instructed to proceed with program execution, **ODT** must reinstate the instruction at **X**, adjust the return **PC** saved on the stack (what adjustment will be needed?), and return with **RTI**.

After a program reaches a breakpoint, it is often desirable to watch what the code does one instruction at a time. **ODT** provides such a single-step feature but does not use **BPT** to achieve it; indeed, it would be cumbersome coding to place and remove a **BPT** for each single-instruction step. Instead, **ODT** uses another way of causing a trap through location **14**: setting the *T bit*, which is bit 4 in **PS**. The *T bit* is set by letting an instruction that loads a new **PS** operate with a word in which bit 4 has been set. Thus **ODT**'s response to a request for a single instruction step could conclude with

```
BIS      #20,2(SP)
RTI
```

The hardware is so designed that, although **RTI** has now set the *T bit*, the CPU has not "noticed" it yet; so the CPU will proceed to fetch the next instruction as usual. Since **RTI** has restored control to the user program, this next instruction is just the one that was to be executed in single-step mode. The instruction is carried out, and now the CPU "notices" that the *T bit* in **PS** is set and traps through location **14**; this is called a *trace trap*.

EXERCISES:

- (i) Should **ODT** do anything to **2(PS)** when it gains control after a trace trap?
- (ii) What instructions besides **RTI** can set or clear the *T bit*?
- (iii) What happens if the instruction being single stepped sets or clears the *T bit*?

3.5 STRUCTURED ASSEMBLY

In this section we consider some of the facilities offered by the assembler to aid in structuring programs.

Macros As we observed in Section 3.3, the PDP-11 lacks a byte-oriented version of the **ADD** instruction. There is, of course, no great difficulty in writing our own code to add data stored in bytes. Suppose we want to perform the operation that

```
ADDDB      X,Y
```

would perform if the **ADDDB** instruction existed. A reasonable approach is to move the byte contents of **X** and **Y** into temporary locations, add the contents of these temporary locations together, and move the byte result into **Y**.

This is not quite adequate, because it does not leave the condition codes properly set. It is rare for a programmer to know with utter certainty that an addition cannot overflow under any circumstances, so our code to simulate **ADDDB** should not deny us the opportunity of checking the **V** bit. However, if we move our byte data into the low bytes of our temporary locations, the subsequent addition will not have its result reflected in the state of the condition codes. To set the codes properly in the first instance, our data must be moved into the high bytes of words that have their low bytes null.

```

MOV B      X,L1+1
MOV B      Y,L2+1
ADD        L1,L2
...
...
L1:        .WORD      0
L2:        .WORD      0
```

In this case, it is more tedious to use registers for the computation (but write the code for practice, remembering the sign-extend property of **MOV B** to a register).

The **ADD** instruction leaves the result we want in **L2+1** and the condition codes properly set. But we have no way of transferring the data to **Y** without clearing the **V** bit! Check for yourself that **MOV B L2+1,Y** leaves the other three codes unaffected in this case. Thus a little trickery is needed, using a **BVS** instruction to ensure that, if **V** is set at this stage, a **SEV** is performed *after* the data byte has been moved to **Y**.

EXERCISE: Complete this “**ADDDB**” routine.

While it may be helpful to have a routine to perform the mythical **ADDDB** operation, it would be tedious to have to include it in a program whenever such a computation is required. Of course, we could write it as a subroutine, but

```
JSR        R5,ADDDB
.WORD      X,Y
```

is less pleasing than the elusive

```
ADDDB      X,Y
```

The assembler, as we observed in Section 1.4, allows programmers to create their own instructions, or *macros*, and invoke them with a single assembler language statement. We have seen several examples of system macros; the assembler directive `.LIST ME` will have shown you that the assembler *expands* macros into their constituent code. We shall now see how to write a macro `ADDB` so that the command

```
ADDB      X,Y
```

may be included in a program exactly as if `ADDB` were an assembler language instruction after all.

Before such a *macro call* may be issued in a program, there must be found, preceding the call within the program, a *definition* of the macro. The definition has as its first line a `.MACRO` directive. here we would have

```
.MACRO    ADDB      X,Y
```

giving, as arguments to the `.MACRO` directive, first the name we have chosen for the macro and then a list of the variables on which our routine is to operate. There must be a *separator character* between variables, and between macro name and variable. In this setting, spaces, tabs, and commas are all permitted as separators. By using commas to separate variables but a tab to separate the macro name from the variables, our `.MACRO` directive helpfully displays the macro call as it will later be made. In the present case, our routine is to operate on the two byte locations whose contents are to be added, and we are calling them `X` and `Y`.

The macro definition continues, after the `.MACRO` directive, with the code that the macro call is to represent (the *body* of the macro). Here this is just the code you wrote for the preceding exercise to complete the “`ADDB`” routine. Following this, the final statement of a macro definition must be the directive

```
.ENDM
```

to inform the assembler that the macro definition has now finished. Optionally, the `.ENDM` directive may have the name of the macro as argument

```
.ENDM      ADDB
```

Including this can help you to see where you are when writing a program.

Thus the definition of the system macro `.TTINR` referred to in Section 3.4 is

```
.MACRO    .TTINR
          EMT      340
.ENDM
```

This presents `.TTINR` as a routine to read a character into a preselected location, `R0` (the `EMT` coding takes care of this), so there are no further arguments to the `.MACRO` directive.

Macro Arguments Our macro `ADDB` does have further arguments, the locations `X` and `Y`. It is important to understand the role played by these symbols, which are called *dummy arguments*.

The function of dummy arguments is to establish the syntax of the macro call.

Our `.MACRO` directive tells the assembler to expect the macro `ADDB` to be called with two arguments, and that in the macro body, which follows, `X` is to “mean” the first argument given when the macro is called and `Y` the second.

Suppose now that somewhere in the program following the definition of macro ADDB, we instruct

```
LABEL:      ADDB      @MEM,WRD+1
```

where MEM and WRD are locations defined elsewhere in the program. The assembler relates @MEM, the first argument in the macro call, to X, the first dummy argument in the .MACRO directive. Therefore, when it expands the macro call at LABEL, *it replaces all references to X in the macro body with references to @MEM*. Since the first line of the macro body was MOV B X,L1+1, the first line of the expansion will be

```
LABEL:      MOV B     @MEM,L1+1
```

Similarly, the assembler will substitute WRD+1 whenever Y occurs in the macro body. Thus, the line after LABEL will be

```
MOV B       WRD+1,L2+1
```

The arguments in the macro call may be any expressions with which the assembler can deal.

Observe that in the macro call itself we make no reference to the names chosen to represent the dummy arguments; the assembler performs the substitutions automatically. Indeed, our ADDB call *cannot* refer to X and Y, since they have not been defined as locations within the program. There is no reason for the assembler to set aside locations for dummy arguments, so the names X and Y are wholly meaningless outside the macro definition. It follows that the same names can be used without ambiguity in various macro definitions and also, if properly defined within the program, in the ordinary way as location names.

EXERCISE: Create ADDB as a macro. Write a program that includes an ADDB call. Study listings of your program both with and without a .LIST ME directive.

Macro Labels A macro may be called more than once in a program. The assembler will perform the same operation each time, replacing the macro call with the entire code of the body of the macro. The code will contain the arguments of the particular call in place of the dummy arguments. Thus, one way of defining the system macro .TTYIN to allow any location to be chosen as destination for the character would be

```
.MACRO      .TTYIN      X
             EMT         340
             BCS         .-2
             MOV B       R0,X
.ENDM
```

After this, successive calls

```
.TTYIN      MEM
...
...
.TTYIN      WRD
```

would expand into

```

      EMT      340
      BCS      .-2
      MOVB     R0, MEM
      ...
      ...
      EMT      340
      BCS      .-2
      MOVB     R0, WRD

```

Note that if we include a definition of `.TTYIN` in our program, there is no need for an `.MCALL` directive.

Suppose now that, paying the utmost respect to our earlier advice to avoid expressions involving `.` for the destination in a branch instruction, we instead define the macro as

```

.MACRO      .TTYIN      X
L1:          EMT      340
            BCS      L1
            MOVB     R0, X
.ENDM

```

Successive calls on `.TTYIN` would then result in two different locations both being given the name `L1`, which would cause an error when the assembler tried to encode the second call.

The assembler will accept the task of creating different label names each time the macro is called, as long as it is told which labels it should do this for. The labels for which these *local symbols* are required must be declared in the `.MACRO` directive as arguments, each preceded by `?`. It improves clarity to separate them from the list of “real” dummy arguments with a tab.

```

.MACRO      .TTYIN      X          ?L1

```

This is the only action the programmer need take. Neither the macro body nor the form of the macro call should be changed.

The assembler will create local symbols for these labels, using the familiar `n$` form but starting from `64$`. Note that just as with programmer-created local symbols, the numbers preceding the `$` sign are decimal.

Figure 3.7 shows a listing of a program section containing an operative version of the `ADDB` macro, together with two somewhat improbable calls on it to illustrate argument substitution. Note that because the argument `<WRD - MEM>&'ZX` contains angle brackets, the whole argument must be enclosed within a further pair of angle brackets; otherwise the assembler will regard the `>` as concluding the argument and the `&'ZX` as an improper label to substitute for the dummy `L1`.

Observe that the program line numbers indicate only programmer-written instructions, not assembler-substituted code. Since the macro definition itself generates no code, it may appear anywhere in the program—as long as it precedes the first call on it.

Note how we manage to put a macro label on the instruction following the macro. Remember that the assembler ignores blank lines, as you can see from the second column of the listing.

MACTST MACRO V03.01 5-JAN-79 09:52:22 PAGE 1

```

1          .TITLE  MACTST
2          .LIST   ME
3          .MACRO  ADDB      X,Y      ?L1,?L2,?L3,?L4
4              MOVB      X,L1+1
5              MOVB      Y,L2+1
6              ADD       L1,L2
7              BVS       L3
8              MOVB      L2+1,Y
9              BR        L4
10         L1:      .WORD   0
11         L2:      .WORD   0
12         L3:      MOVB    L2+1,Y
13         SEV
14         L4:
15         .ENDM
16
17 000000      START:  ADDB    @MEM,<<WRD-MEM>&"ZX>
000000      117767 000124 000027      MOVB    @MEM,64$+1
000006      116767 000002 000023      MOVB    <WRD-MEM>&"ZX,65$+1
000014      066767 000014 000014      ADD      64$,65$
000022      102406      BVS     66$
000024      116767 000007 000002      MOVB    65$+1,<WRD-MEM>&"ZX
000032      000406      BR      67$
000034      000000      64$:     .WORD   0
000036      000000      65$:     .WORD   0
000040      116767 177773 000002      66$:     MOVB    65$+1,<WRD-MEM>&"ZX
000046      000262      SEV
000050      67$:
18 000050      ADDB    MEM-WRD,@MEM
000050      116767 177776 000027      MOVB    MEM-WRD,68$+1
000056      117767 000046 000023      MOVB    @MEM,69$+1
000064      066767 000014 000014      ADD      68$,69$
000072      102406      BVS     70$
000074      116777 000007 000026      MOVB    69$+1,@MEM
000102      000406      BR      71$
000104      000000      68$:     .WORD   0
000106      000000      69$:     .WORD   0
000110      116777 177773 000012      70$:     MOVB    69$+1,@MEM
000116      000262      SEV
000120      71$:

```

Figure 3.7 A listing of the ADDB macro.

EXERCISES: Write macros to meet the following specifications.

- (i) **SWAP X,Y**
interchanges the contents of locations X and Y; create a byte version also.
- (ii) **FILCMP X,Y**
compares word by word the contents of two blocks that start at X and at Y and end in each case with the first null word encountered. If the blocks are identical in length and contents, the macro is to delete the entire block, starting at Y.
- (iii) **IREAD N,Y**
reads N integers from the terminal into locations starting at Y.
- (iv) **PRINT X**
sends to the terminal ASCII text in bytes starting at location X and continuing until a null byte is encountered. Decide whether you want a call to print out data starting from MEM to be **PRINT MEM** or **PRINT #MEM**.

In Exercises iii and iv you will have created *nested* macros. Your macro **IREAD** will have included a call on the system macro **.TTYIN**, and **PRINT** will have included a call on **.TTYOUT**. One macro may call another, with nesting to any depth, as long as each macro is defined in the program before its first call.

As the example in Figure 3.7 shows, long macros can easily generate extravagantly long programs. This is avoided by writing the macro as just a calling sequence to a subroutine that performs the task (how does this avoid the problem?).

```
.MACRO      ADDB
            JSR      R5,XADDB
            .WORD    X,Y
.ENDM
```

We personally prefer to distinguish the subroutine from the macro call (here by prefixing **X**), but the assembler does not require this. Note that system macros use this technique; the macro is merely a calling sequence for an EMT instruction.

Radix Control Macros that may be used in various programs must take care that nothing in a program can affect their operations adversely. System macros guard against the **.RADIX** assembler directive, which takes as argument one of the (decimal) numbers 2, 4, 8, 10 and causes the assembler to regard all numbers in the instructions that follow as being in the stated radix (base). Thus

```
.RADIX      10
CMP         #1000,MEM
```

compares the contents of **MEM** with *decimal* 1000.

If a **.RADIX 10** had been issued, a **.TTYIN** monitor call would have its **EMT 340** instruction assembled with 340 regarded as a decimal number; this would transfer control to the wrong EMT routine. The macro takes the precaution of resetting the radix to eight, just for the 340 in the EMT instruction. The symbol **^** (up-arrow, not CONTROL) followed by a letter specifies a radix valid for the following number (or expression, in angle brackets). The letter **B** (binary), **O** (octal), or **D** (decimal) may be used.

```
EMT         ^O340
```

Conditional Assembly The RT-11 system macro **.TTYIN** is defined as

```
.MACRO      .TTYIN      CHAR
            EMT         ^O340
            BCS         .-2
            .IF NB <CHAR>
            .IF DIF <CHAR>, R0
                MOVB     %0,CHAR
            .ENDC
            .ENDC
            .ENDM
```

(some RT-11 versions may differ slightly), with the instruction **MOVB %0,CHAR** enclosed within two levels of directives to the assembler. The function of these directives is to ensure that the instruction is not assembled under certain circumstances when it would be superfluous.

The **.IF** directive is followed by a condition under which assembly is to take place and then by the expression(s) to be tested to see if the condition is met. Condition and expression may be separated by one or more spaces, a tab, or a comma. Expressions must be separated by a comma.

We have here two `.IF` conditions designed specifically for use with macro arguments. When a macro argument is tested in one of these `.IF` directives, it should be enclosed within angle brackets.

The `.IF NB` directive tells the assembler to assemble instructions that follow when it encodes a macro call, only if the argument given in the call is NonBlank. A call `.TTYIN MEM` has the argument corresponding to `CHAR` nonblank, so the `.IF NB <CHAR>` condition is satisfied and the assembler proceeds to the next line, which we consider later.

Suppose, however, that the call is `.TTYIN`. This does have a blank in the argument field corresponding to `CHAR`, so the assembler will refrain from assembling the code that follows until the `END Conditional` directive `.ENDC` *that is paired with the .IF directive* is encountered. Here we have nested `.IF` directives, each paired with its `.ENDC` directive like left and right parentheses. The `.IF NB` directive is paired with the second `.ENDC` and the `.IF DIF` directive with the first `.ENDC`.

Thus a call `.TTYIN` results in the assembler producing

```
EMT      ^O340
BCS      .-2
```

Remember that this `EMT` routine reads a character into `R0`, so `.TTYIN` is interpreted as `.TTYIN R0` without encoding the superfluous instruction `MOVB R0,R0`.

The `MOVB` instruction is superfluous not only with the call `.TTYIN` but also with `.TTYIN R0`. The argument field here is nonblank, so the `.IF NB` directive will not prevent assembly of the `MOVB` instruction. This is the function of the `IF DIfferent` directive `.IF DIF`, causing the assembler to encode the following conditional block only if the two expressions given as arguments to the directive are different. The two arguments may both be macro arguments, or a macro argument and an expression to check against it, as previously shown. According to the manuals, here again any macro arguments must be enclosed within angle brackets; however, we have used several assemblers with which this proved to be unnecessary.

Thus suppose we are writing a macro `SWAP` to interchange the contents of two locations.

```
.MACRO    SWAP      X,Y
```

If the macro is called with the same location as both arguments,

```
SWAP MEM,MEM
```

there is no need to do anything. We can either rely on the good sense of future users of the macro not to use such a call, or we can inhibit the assembler from doing anything if it is made, by continuing with

```
.IF DIF <X>,<Y>
```

as an introduction to the macro body. The `.ENDC` for this conditional will then immediately precede the `.ENDM` that terminates the macro definition.

If the macro were to use `R0` as a temporary location for use during the swap, it could continue with

```
MOV      X,R0
MOV      Y,X
MOV      R0,Y
```

or, for maximum economy,

```
.IF DIF      <X>,R0
              MOV      X,R0
.ENDC
              MOV      Y,X
.IF DIF      <Y>,R0
              MOV      R0,Y
.ENDC
```

For conditions expressible in one line, the Immediate IF directive `.IIF` may be used; this does not have an `.ENDC` statement. The whole macro has now become:

```
.MACRO      SWAP      X,Y
.IF DIF      <X>,<Y>
.IIF DIF      <X>,R0      MOV      X,R0
              MOV      Y,X
.IIF DIF      <Y>,R0      MOV      R0,Y
.ENDC
.ENDM
```

The `.IF DIF` conditional has a complementary directive: assemble IF arguments are IDeNtical, `.IF IDN`. The complementary directive to `.IF NB` is `.IF B`, IF Blank. These conditionals all regard a macro argument merely as a character string. `.IF B` and `.IF NB` test whether there are any characters at all in the string. `.IF IDN` and `.IF DIF` test whether two strings have the same characters. Thus, if `MEM` is location relocatable 100 and `WRD` is relocatable 200, a call `SWAP MEM+100,WRD` would result in assembly of all three instructions. As *character strings*, `MEM+100` and `WRD` are quite distinct, even though as *expressions*, they are both equal to relocatable 100.

EXERCISE: How does the assembler encode

- (a) `.TTYIN %0`
- (b) `REG0 = %0`
`.TTYIN REG0`

Expression Value Conditionals We can make assembly conditional on the inequality of our arguments as expressions by using the IF Not Equal to zero directive `.IF NE`.

```
.IF NE X-Y
```

Note that no angle brackets are needed in conditionals testing expressions. Observe also that `.IF NE` tests only *one* expression, comparing it with zero.

Thus, instead if `.IF DIF <X>,<Y>`, our `SWAP` macro might use the `.IF NE X-Y` conditional. Suppose again that `MEM` is relocatable 100, and `WRD` is relocatable 200. Now the call

```
SWAP      MEM+100,WRD
```

has recognizably equal arguments. Note that the arguments in

```
SWAP      WRD-MEM,MEM
```

are unequal; the first is the absolute number 100, the second is relocatable 100. (What if this call appeared in an `.ASECT` section?) Indeed, for relocatable `MEM` this call will be regarded by the assembler as an error, since it computes `WRD - MEM - MEM`; this is neither relocatable nor absolute, hence is not permitted as an expression.

Note that a call `SWAP MEM,R0` would also cause an error, because the assembler would try to compute `MEM - R0`; since registers do not have addresses on the PDP-11, this is not a permitted expression.

There are `.IF` expression-testing directives to check whether an expression, compared to zero, is: Not Equal, `NE`; Greater Than, `GT`; Less than or Equal, `LE`; Less Than, `LT`; or Greater than or Equal, `GE`.

Of course, there is no way at all in which an assembler directive can inhibit assembly according to the value of the *contents* of a location (why not?).

Branched Conditionals We may not always want simply to expand part of a macro if a condition is met, ignoring it otherwise; we may want to take alternative courses of action, depending on whether the condition in an `.IF` directive is met.

Suppose, for example, that we want a macro `STORE`, referencing one argument, to respond to `STORE MEM`, where `MEM` is a memory location, with the code `MOV #MEM, -(SP)`; but to `STORE R`, where `R` is a register, with the code `MOV R, -(SP)`. Such a macro might be nested inside another macro to deal with the latter's arguments under certain circumstances.

First, we have to be able to recognize that an argument is a register. The assembler directive `.NTYPE` within the body of the macro will tell us the mode and register used to address any argument. The `.NTYPE` directive takes two arguments: a name we want to identify with the information returned by `.NTYPE`, and the argument in which we are interested. We can choose any name, according to the usual rules; it is set equal to the six-bit code giving mode and register of the argument, just as in a direct assignment statement. Thus, after

```
.MACRO      STORE      X
           .NTYPE      A,X
```

the symbol `A`, when referred to within the condition of an `.IF` directive, will be equated to an octal number between 0 and 77, corresponding to the six bits giving mode and register for the argument substituted for `X` in the macro call.

The argument will be a register if and only if its addressing mode is zero: that is, if and only if `.NTYPE` returns a value between 0 and 7. Thus, after

```
           .NTYPE      A,X
. IF EQ A&70
```

code will be assembled only if the argument in the macro call was a register. Thus the next line in the macro body should be `MOV X, -(SP)`. We are now able to write the whole macro.

```
.MACRO      STORE      X
           .NTYPE      A,X
. IF EQ A&70
           MOV         X, -(SP)
. IFF
           MOV         #X, -(SP)
. ENDC
. ENDM
```

The directive `.IFF` is: assemble if the condition tested when the current conditional block was entered is False. Note that `.IFF` does not mark the beginning of a new conditional block; rather, it is *subordinate* to the `.IF EQ` conditional, indicating the code to be assembled if the condition was not met.

- EXERCISES:**
- (i) Rewrite `STORE` using `.IF NE` instead of `.IF EQ`.
 - *(ii) Rewrite `STORE` so that any addressing mode may be used in its argument.
 - (iii) Write a `BMOV` macro, to perform the exact function of `MOVB`, except that `BMOV` to a register will leave the high byte unaffected.
 - (iv) Write a macro to compute the `AND` function of the contents of two locations.

Forward References Problems may arise when a macro references a location that follows it later in the program. The problems and the response to them will differ according to the version of assembler used, making this area rife with pitfalls.

A macro may require certain procedures to be carried out the first time it is called in a program but not thereafter. For example, memory for temporary storage may need to be reserved. A traditional trick for doing this is to use the IF DeFined directive `.IF DF`, which allows assembly only if the symbol given as argument has been defined; the complementary directive is `.IF NDF`. The symbol chosen as argument is one not used elsewhere in the program. Thus, macro `STARTUP` would be called once only by including in the main macro the code

```
.IF NDF      Q
              Q=0
              STARTUP
.ENDC
```

`Q` is undefined on the first call of the main macro; but this call defines it, as being equal to zero, and so on the next call the `.IF NDF` directive inhibits assembly.

Some versions of MACRO-11, however, will “notice” the direct assignment statement `Q=0` within the macro body, and enter `Q` in the program’s symbol table, before the stage at which conditional assembly decisions are made. `Q` would then be a defined symbol, even at the first call of the main macro, so `STARTUP` would never be called at all.

With such assemblers `Q` could be defined within a macro that the program can search for in a library. Thus the RT-11 system macro `.MACS` performs certain procedures needed by other system macros, and its code includes the direct assignment statement `..V1=3`. Other RT-11 macros include the code

```
.IF NDF      ..V1
              .MCALL      .MACS
              .MACS
.ENDC
```

The question of whether a symbol is defined or not arises in a somewhat different context. A favorite textbook problem is to write a macro, taking a location as argument, to generate a `BR` instruction if the location is within range and a `JMP` if it is not.

```
.MACRO      J          X
  .IF condition
            BR          X
  .IFF
            JMP         X
.ENDC
.ENDM
```

(What should *condition* be?) Suppose first that we have

```

LABEL:      ...
            ...
            ...
            J          LABEL
```

There is no difficulty here; the assembler checks whether the expression `LABEL - .` is of sufficiently small magnitude for a `BR` and encodes accordingly.

Suppose, however, that we have

```

            J          LABEL
            ...
            ...
LABEL:      ...
```

The assembler may respond in either of two ways, depending on the version being used. If the assembler is so constructed that it finds itself called on to encode the macro before reaching `LABEL` and inserting it in the symbol table, it will regard `LABEL` as an undefined symbol. This will be treated as an error. With such an assembler, the `Q` trick just discussed would work.

Alternatively, the assembler may be equipped to recognize that `LABEL` has been defined. However, since the assembler has not yet encoded the instructions lying between `.` and `LABEL`, it cannot know the *value* of `LABEL`. Thus it cannot encode the address field of `JMP` or the offset of `BR` at this stage. Indeed, the assembler may be unable to determine whether a `BR` or `JMP` will be needed.

There are various techniques of assembler design (beyond the scope of this book) to deal with this problem of forward referencing of labels by macros. As a result of these techniques, a macro may reference forward locations by a `MOV` (as we have already seen with `ADDB`), a branch instruction, and so forth. A forward `JMP`, however, will result in different values being ascribed to the destination at different stages of the assembly process; this is called a *phase error* and is marked with the code `P` in an assembly listing.

4 Peripheral Devices

4.1 TERMINAL AND CONSOLE I/O

Until now we have left the task of transferring data between computer and terminal to the operating system, using monitor calls such as `.TTYIN`, `.TTYOUT`, and `.PRINT`. Recall that the expansion of these system macros includes in each case an `EMT` instruction, which forces a jump to a special routine handling the I/O. The general user in a timesharing system must rely on these traps to monitor controlled routines, as such systems prevent direct access to the I/O mechanism by users without special privileges.

On a standalone system, however, direct control of I/O is possible and is the subject of this and the next sections. Whether or not such facilities are presently available to you, you should study these sections in their entirety, as they will enhance your understanding of the PDP-11 structure.

Let us return to our very first program and learn how to print the letter **B** at the terminal, but this time without the monitor's assistance. It is the job of the program to deliver data to the point at which the mechanics of the terminal can take over. This point is an electronic storage device within the terminal, called the *terminal printer buffer*. Data in the printer buffer are treated by the terminal as ASCII code and appear as the appropriate characters typed on paper or displayed on screen.

The terminal printer buffer is assigned an address, just like a memory location. Data can be moved into it using normal assembler language instructions and any form of addressing. The address of the terminal printer buffer is prescribed by the hardware as `177566`. You may find, when using the switches as next discussed, that you need to set them to `777566`, or even to `17777566`, to deposit directly into the terminal printer buffer. However, `177566` is still the address to use for the buffer in a program, including a program to be loaded via the switches. The reason for this disparity will be considered in Section 4.5

The buffer contains eight bits usable by the programmer and should be addressed as a byte. So our program is

```
TPB = 177566
START:  MOVB    #102,TPB
        HALT
        .END    START
```

The mnemonic `TPB` is customary for the terminal printer buffer.

This section's philosophy of independence of the monitor prompts us to stop the CPU with `HALT` rather than returning control to the monitor with `.EXIT`. Such a simple program can even be loaded and run without the aid of an operating system.

The Console Switches Most models of PDP-11 have an operator's console replete with lights and switches through which the system may be controlled. Let us consider how to run the preceding program by using the console switches. Since we have no assembler, we must encode it ourselves. While writing the code, it is important to remember that we have no linking loader either, so unless we want the extra task of relocation, we should write position-independent code. Thus, stripped of instructions to the assembler, our program and its octal code are

112737	MOVB	#102,@#177566
000102		
177566		
000000	HALT	

Operating systems usually load user programs starting at location 1000. With absolute control we can load at any location we please; nevertheless, it is as well to avoid the various trap vectors and to leave room for the stack, so we, too, shall start loading at location 1000.

We turn the computer on—but not with the on/off switch set to “lock,” which disables the console switches—and set the console switches to indicate 1000. The switches, numbered from 0 on the right, represent bit positions (the most sophisticated PDP-11s boast octal displays). To indicate 1000 (octal), we just raise switch 9 (decimal) into the up position to register a 1 in the corresponding bit. We now press the Load Address switch (denoted here by L) and can see address 1000 show up in binary notation in the address register lights (a light on means a 1).

We can proceed to load the program. Set the switches to represent 112737, the first word to be loaded, and raise the Deposit (D) switch. The new contents of location 1000 will be displayed in the lights of the data display register.

The D switch has an automatic stepping feature. Raising it deposits data into the current address and increments the address register by 2. Thus the switches can be set to indicate the next word of the program coding, D raised again, and so on. When loading is finished, L can be used to set the address register to 1000 again, and the coding can be checked by using the Examine (E) switch; this also has an automatic stepping facility.

All that remains is to use L to reset the address register to 1000; check that the switch to stop the processor does not indicate HALT and flip the Start (S) switch. Operations will start at location 1000, B will appear, and the CPU will stop.

Some consoles provide special switch settings to examine, or deposit into, the registers. The registers will normally be accessible as if they were at “addresses” 177700 for R0 to 177707 for PC. Note that an increment of 1 in the switch setting yields the next register. The D and E switches offer automatic stepping through the registers on some CPUs, not on others. Note also that these “addresses” are meaningless in a program.

Not all PDP-11s are equipped with console switches. Some of the development work for this chapter was carried out on a PDP-11/04 with a *console emulator* instead of switches. This is a hardware device, invoked by pressing the BOOT switch on the 11/04 (the reason for the name will become clear later). The console emulator is a program but, rather than having to be loaded into memory, it is permanently built into a special block of *read only memory* (ROM).

The console emulator routine sets up communication between the CPU and the terminal and then accepts commands to perform the various functions of the console switches. The 11/04 emulator prompts with \$ to indicate its readiness to receive a command. To load the same program as before, the user would type

L 1000←

and receive another \$ prompt, then type

D 112737↵

then, after the \$, type

D 102↵

omitting leading zeros; and so on.

Typing E followed by a space (not ↵) has the same effect as pressing the E switch. Typing S ↵ starts the program at the previously loaded address; note that the ↵ is not echoed at the terminal printer.

A console emulator offers a reasonable solution to the problem of obtaining control over a computer that lacks switches. It is, however, impossible to avoid a certain sense of distance from the machine when working within the confines of even such a simple operating system.

The emulator routine is used to invoke other operating systems. On the 11/04 we used the RT-11 operating system on a floppy disk. System syntax required the command DX ↵ in response to the emulator routine's \$ prompt, whereupon the RT-11 system responded with a series of messages and a . monitor prompt.

EXERCISE: When the emulator routine is halted for any reason and restarted, it prints out the contents of R0, R4, SP, and PC as at the time of the halt. Use this information to devise a two-word sequence to be loaded at the end of your own programs so that control returns to the emulator, analogous to a monitor .EXIT call.

Sequential Character Output It might seem that a program with more than one character to output should merely repeat the process just described. We would then have something like

```

      MOV      #MESSAGE,R1
LOOP:  TSTB     (R1)
      BEQ      DONE
      MOVB     (R1)+,TPB
      BR       LOOP
      ...
      ...
MESSAGE: .ASCIZ  /TESTING OUTPUT/
```

(How does what we are trying to do differ from the .PRINT monitor call in its intended effects?) We would use a similar looping technique to output the results of computation. Note that since we are outputting data stored in successive bytes at MESSAGE, we get correct autoincrementing with the MOVB instruction.

This simple procedure does not work for a reason that exerts a profound influence on all I/O programming considerations: the CPU can send data to the terminal printer buffer very much faster than the terminal can print it. The printer buffer can hold only one character and, until the current character is printed, any attempt to send a new one must fail. Consequently, if the CPU sends too rapid a stream of characters, many will be lost. You should expand the previous routine into a program and see this effect for yourself.

To ensure that a character gets printed, the CPU must check before sending it that the printer buffer is ready to receive it. This check is made by referring to the terminal *printer status register*, customarily given the mnemonic TPS. The status register, like the printer buffer, has an address; TPS

is one word address below **TPB**. Bit 7 of **TPS** is the “ready” bit; this bit is clear when a character is being processed in the printer buffer and is set when the buffer is ready to receive a new character. Thus a program must wait until bit 7 of **TPS** is set to 1 before transmitting a character to **TPB**. Conveniently, bit 7 is the sign bit of the low byte in a word. Our printing loop has now become

```

                TPS = 177564
                TPB = 177566
                ...
LOOP:          MOV      #MESSAGE,R1
                TSTB    (R1)
                BEQ      DONE
1$:            TSTB     TPS
                BPL      1$
                MOVB    (R1)+,TPB
                BR       LOOP

```

Note that bit 7 of **TPS** is set or cleared by the terminal electronics only. Any attempt by a program to write into this bit is ignored; it is a *read only* bit.

- EXERCISES:**
- (i) Expand the preceding routine into a complete program to print out a message.
 - (ii) Write a program to print out the contents of a given location as a decimal number.
 - (iii) Estimate the speed of the CPU relative to the printer by including in the loop at **1\$** an instruction to count passages through the loop.

Character Input The terminal keyboard is wired to the computer in a way that allows addressing of its buffer and status register, just as with the printer buffer. These two locations generally are:

```

TKS = 177560      ;keyboard status
TKB = 177562      ;keyboard buffer

```

A character typed at the terminal keyboard is processed by the terminal into its ASCII code within the keyboard buffer. When the terminal is working in local mode, the character is transmitted from the keyboard buffer to the printer buffer, enabling the terminal to be used as a typewriter. When the terminal is on line to the computer, however, there is no direct connection between **TKB** and **TPB** (or, if there is one, it can be switched off). In this state the terminal has finished with an input character once its ASCII code is stored in **TKB**. The terminal will set bit 7 of **TKS** to 1 to indicate that a character is available in **TKB**. Bit 7 of **TKS** is the “done” bit, analogous to the ready bit in **TPS**. The character is held available for a fraction of a second only; if the CPU does not pick it up during this time it is lost, and the done bit is cleared to indicate willingness to receive a new character.

Thus the CPU must be coordinated with terminal input just as with output. As soon as a character is read from **TKB**, the terminal clears the done bit in **TKS**, ready for the next input character. Once again, only the sign bit in the low byte at the status register need be tested. Thus a loop to read a string of characters into a block of bytes pointed to by **R1** is as follows.

```

LOOP:          TSTB     TKS
                BPL      LOOP
                MOVB    TKB,(R1)+
                BR       LOOP

```

Since the loop to test TKS will be performed many times, we can improve the routine considerably by suitable addressing.

```

      MOV      #TKS,R2
LOOP: TSTB     (R2)
      BPL      LOOP
      MOVB     2(R2),(R1)+
      BR       LOOP

```

However, for clarity of exposition, this approach will not be incorporated in the examples.

A problem with this routine is that there is no way to end it. If we stop typing characters the program will just settle into the two-line loop at LOOP. We might want to let $\leftarrow$ terminate input. Assuming that the $\leftarrow$ is to be part of the data stored, we should check each character for LINE FEED after storing it. This would seem easy enough, replacing the last two instructions in our loop with

```

      MOVB     TKB,(R1)
      CMPB     #12,(R1)+
      BNE      LOOP

```

However, some systems operate with bit 8 of TKB set; others use it as a parity bit. (You need not worry about bit 8 when transmitting ASCII coded data, as the terminal printer ignores it.) Thus LINE FEED might be stored as 212, with bit 8 set, and so the desired comparison would not take place. One way of dealing with this is to clear bit 8 of the stored data before making the comparison with #12, by BICB #200,(R1). (Why have we not needed to clear bit 8 when inputting a character until now?)

A program to read characters from the keyboard might not work properly if it is run while a monitor is resident in memory. So if possible, run the program without the monitor. As an alternative to managing without the monitor, you can have your program clear bit 6 of TKS before prompting for input. We shall explain the significance of this in the next section. Your program should set this bit after all input has finished.

- EXERCISES:**
- (i) Write a program to read characters typed in at the terminal into a block of bytes starting at location 2000, terminating when a $\leftarrow$ is encountered.
 - (ii) Load your program “by hand”—that is, using the switches or emulator—and start it. Check that:
 - (a) You do *not* see the characters you type appear at the terminal.
 - (b) All the characters you type are stored by the program.
 - (iii) Amend the program so that the characters *do* appear (echo) as you type them.
 - (iv) Make # into a rubout key, echoing the character rubbed out and deleting it from storage.

Bootstraps We explained earlier that the starting procedure permitted by a console emulator depends on circuitry that automatically invokes a program enabling the CPU to communicate with the terminal. The essential problem in starting a computer is to input the program that enables the computer to perform input; this is a paradox, insofar as any solution appears to depend on the prior existence of a solution. An analogy has traditionally been drawn to an endeavor to lift oneself up by one’s own bootstraps; the analogy is less than persuasive, but the terminology has stuck to this “bootstrap problem.” Programs that set up initial communication between the computer and the terminal or other peripheral device are called bootstrap programs. Some PDP-11s require that they be loaded by hand (*software* bootstrap), via the switches; in this case the operator’s manual will list the code and where it is to be deposited. Fortunately, however, most have bootstrap programs built in and ready for loading at the touch of a single switch (*hardware* bootstrap); the console emulator incorporates such a routine.

If a software bootstrap has to be used, it should be as short as possible to minimize the danger of errors in the loading process. Consequently, software bootstraps provide excellent examples of ingenuity in coding, displaying not only the benefit of economy but also the detriment of opacity.

The usual function of a software bootstrap program is to load and run a longer, more sophisticated loading program that would itself be too tedious to load by hand. The latter program will have been carefully dovetailed to the bootstrap program and will be kept on some permanent storage device. Thus the bootstrap program must first set up communication with the storage device and then run the program on it; this will set up communication with the terminal and probably provide at least rudimentary monitor functions.

When this monitor program is on disk, the bootstrap must reference the disk equivalents of data and status registers. This procedure is too complex to serve as a good illustration here. When the monitor is on paper tape, however, the tape is loaded by the user, and the bootstrap program has only to read it. Although paper tape is currently yielding in popularity to electronic storage media, it is still to be found in many computer installations. Paper tape is passed by the reading machine across a reading head comprising a row of eight sensors, one for each "channel" of the tape. Each sensor can detect whether or not a hole has been punched in the tape at its channel position. Thus, in effect, the paper tape reader reads a sequence of eight-bit bytes; we need not investigate its mechanism further than this. The paper tape reader buffer, **PRB = 177552**, is used like **TKB**. The paper tape reader status register, **PRS = 177550**, is similar to **TKS**. However, bit 1 of **PRS** is the "reader enable" bit, which must be set by the program each time a byte is to be read; reading the byte automatically clears it, and it must be reset for the next byte.

Thus the user should load the paper tape holding the monitor program into the reader, load the bootstrap program through the switches, and start the computer. The monitor will then respond at the terminal. The PDP-11 processor handbook for the 11/03 gives the bootstrap program shown in Figure 4.1 for a machine whose highest memory address below the block normally reserved for peripheral device addressing is **077776**. The translation into assembler language is our own; note that the bootstrap program must be loaded into the addresses specified, and the assembler language statements take this into account.

The program starts by letting **R1** point to **PRS**. At **LOOP**, it tests for input in a familiar fashion and moves a byte to memory from **PRB** when one arrives. Note that **INC (R1)** has previously set the reader enable bit in **PRS**. Then the program branches back to **S1** for more input.

Let us look at the mysterious instructions. At **S1**, observe that the instruction is effectively a data move from location **77752** into **R2**. After that instruction is performed, **R2** contains **352**. Thus the first character to be read is deposited at location **77400(R2) = 77400 + 352 = 77752**. Note that this is the word that originally held the data **352** for moving into **R2**. Note also that the next instruction increments the contents of that location.

Plainly the bootstrap program is self-modifying. The purpose is impossible to fathom, however, without knowing that a program on paper tape is provided with a *leader* of the same byte repeated through some inches of tape. This lessens the risk of physical damage to the program itself and clearly

```

077744 016701 START:  MOV    77776,R1
000026
077750 012702 S1:     MOV    #352,R2
000352
077754 005211         INC     (R1)
077756 105711 LOOP:   TSTB   (R1)
077760 100376         BPL     LOOP
077762 116162         MOVB   2(R1),77400(R2)
000002
077400
077770 005267         INC     77752
177756
077774 000765         BR     S1
077776 177550         .WORD   PRS

```

Figure 4.1 A paper tape bootstrap loader program.

identifies the start of the program as the place where the leader stops. The monitor program here is to have the leader 351. When a 351 byte is read, it is, as we have seen, deposited into location 77752. The contents of that location are then incremented, so when BR S1 is performed, location 77752 contains 352, just as it did originally. Thus the program is not modified by reading a leader byte; nor, therefore, by any number of leader bytes.

Any other byte read from the tape will, however, modify the program. Consider the effect of the first byte that is not 351. When it is read, R2 still contains 352 from the last byte reading loop, and the new byte will still be read into location 77752. Thus, after the branch back to S1, location 77752 contains the new byte plus one; this is then moved into R2. R2 now points to a new location, which determines a new address for the next byte to be read; this address will be the first word of the monitor program.

The monitor program is to be loaded into locations immediately preceding those occupied by the bootstrap program in memory. Suppose that the monitor program is to be loaded starting from location 77600. We must ensure that the first byte after the 351 leader sets up R2 so that the MOVB instruction moves the next byte read to location 77600; that is, R2 must contain 200. Now location 77752 has been incremented before being moved into R2, so the byte loaded into it from the previous read loop must have been 177. Thus the tape sequence 351, . . . , 351, 177 leaves locations 77752 and R2 both containing 200 when the next byte is read.

Observe now that the program has ceased to be self-modifying! Location 77400(R2) is no longer within the program; and each read loop, by incrementing location 77752, just ensures that R2 is incremented so that the next byte read is stored in memory after the previous one. Thus the tape sequence 351, . . . , 351, 177, *program* loads the program into memory starting at location 77600.

The monitor program will have been devised so that its last location immediately precedes the first word of the bootstrap program. However, letting the tape terminate at that point would just leave the bootstrap program waiting for more input; it would not transfer control to the monitor program. The tape must continue and somehow effect a branch to a location within the monitor program.

Eight more bytes will be read from the tape. The first six are just the same as the first six bytes in the bootstrap program, so reading them leaves locations 77744, 77746, and 77750 just as they were before. The bootstrap program has again become self-modifying, but these six bytes have been set up so that no actual modification takes place. The seventh byte will be read into the low byte at location 77752. Note that when the loop to read this byte is entered, location 77752 once again contains 352. The seventh byte will be 373; thus the loop will return to S1 with location 77752 containing 374.

Suppose that control is to be transferred to location 77600, the first word of the monitor program. Then the last byte will be 701. When its read loop is entered, R2 has been set to contain 374, and the byte is destined for location $77400 + 374 = 77774$. So the MOVB instruction alters the instruction BR S1; with the new code 000701, this is BR 77600 (check it).

4.2 INTERRUPTS

Curious things may happen when a program to input characters without monitor assistance is run with a sophisticated monitor in residence. We ran our program to input characters without echo and store them, but this time with the RT-11 monitor loaded. The result was that all the characters *were* echoed, and some, but not all, were stored; nor were those stored the same ones every time the program was run. Furthermore, the monitor tried to interpret the characters input as a command, which gave rise to error messages.

The problem here was that a normal function of the monitor conflicted with the aims of the program. It is part of the monitor's job to deal with keyboard input: echoing it to the terminal printer, stopping a program if it is two ^C, and so forth. Our program did nothing to inhibit the monitor,

which therefore took input from the keyboard before the program could get to it. It would seem that the cycle of communication between CPU and terminal allowed the program to access the same data.

Our earlier discussions of the operation of the CPU have given no indication of how the monitor obtains control of terminal input. We have described the CPU as performing instructions in sequence, according to the contents of PC. This does not explain how a program running in a perpetual loop can be stopped by typing ^C twice. Plainly, finding that an input character is ^C sends the monitor into a particular routine; how, though, does the monitor program ever gain control of the CPU to read the character in the first place?

The UNIBUS Our answer requires some understanding of the way in which the various parts of a PDP-11 system are connected. Such a system consists of a CPU, memory, terminal keyboard and printer, storage devices such as disks or tapes, and possibly other peripheral devices such as clocks, display terminals, or high-speed printers. It would be impossible to operate a computer system without an electronic communication line, or *bus*, between each peripheral device and the CPU. The PDP-11, instead of having a separate data bus for each device, has one data bus common to all devices. It is called the UNIBUS and is the backbone of a PDP-11 system.

The CPU, memory, and all peripheral devices connect into the UNIBUS. Devices can use the UNIBUS to communicate with the CPU and also directly with each other; thus devices (such as certain disk and tape units) able to send and receive data without CPU assistance do not need to tie up the CPU. A diagram of the PDP-11 system structure appears in Figure 4.2.

The UNIBUS identifies the source and destination of data by its address; these are the same addresses that the CPU, communicating through the UNIBUS, uses. Memory, as we have seen, occupies many thousands of UNIBUS addresses. The terminal printer, as we have also seen, occupies just two UNIBUS addresses: one for status and one for data.

Our program instructions to transmit data between the CPU and memory work because, when the CPU performs them, it has *control* of the UNIBUS. *Only one device at a time can control the UNIBUS.* The device in control can use the UNIBUS to send data to or fetch them from any other device. Obviously, not all devices are equipped to perform such data transmission.

Normally, the CPU has control of the UNIBUS. Other devices may, however, request control. For example, pressing a key on the terminal keyboard causes the terminal to send a signal to the CPU along a special *bus request* line within the UNIBUS. A bus request indicates a desire to control the UNIBUS and to obtain services from the CPU. The CPU will, in any case, finish the instruction it is currently performing. It then checks whether the device (in this case the terminal keyboard) sending the bus request has sufficient *priority* to be granted control of the UNIBUS; we shall discuss this matter shortly but, for now, let us assume that the terminal keyboard has sufficient priority and so obtains control of the UNIBUS.

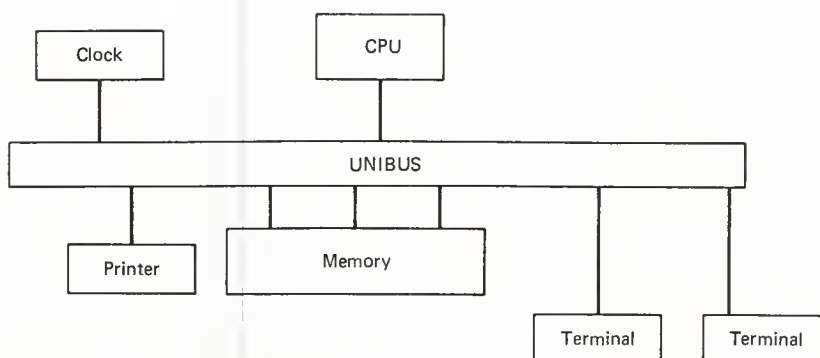


Figure 4.2 A PDP-11 system.

Having gained control of the UNIBUS, the terminal keyboard uses it to transmit an *interrupt request* to the CPU. This is a signal that provokes a special response from the CPU to *service* the request. However, the CPU first checks that the device sending the interrupt request has been *enabled* for interrupts; we shall also discuss this later, assuming for the moment that the terminal keyboard has been enabled for interrupts. The interrupt request sent by a device to the CPU is accompanied by a *vector*, or directive to a memory location: the terminal keyboard transmits with its interrupt request the information "memory location 60." This choice of location is determined by the hardware construction of the connecting device, or *interface*, between the terminal and the UNIBUS. The CPU now sends an acknowledgment signal to the terminal keyboard, which returns control of the UNIBUS to the CPU.

The CPU now performs its response to the interrupt. This response first puts the processor status word and the program counter onto the system stack. Note that the processor status word, **PC**, is referenced as location 177776 (subject to the variations for eighteen- and twenty-two-bit addresses that we noted in the last section). Thus the entire state of affairs in the CPU when the interrupt occurred is saved. The CPU then, treating the address transmitted as the first word of a two-word vector, loads **PC** with the contents of the first word and **PS** with the contents of the second. Observe that the entire response, including the choice of addresses from which **PC** and **PS** are loaded, is prescribed by the *hardware*.

For the terminal keyboard, this has the same effect as the sequence

```
MOV    PS, -(SP)
MOV    PC, -(SP)
```

followed by

```
MOV    60, PC
MOV    62, PS
```

The hardware response to the interrupt is now complete, and the CPU returns to its business of performing instructions. It does not, however, return to whatever it was doing before the interrupt occurred, because **PC** has been changed. Location 60 will have been preset, by the monitor if there is one, otherwise by the user, to point to a routine for handling the input character. This routine is called the *interrupt service routine* for the terminal keyboard. The interrupt service routine may be written to return control to the main program after dealing with the character, but this is not required. Location 60 may be set to point to any routine that the user cares to write. Since the interrupt procedure uses the hardware stack, **SP** must be preset to point to the highest address in the memory block reserved for the stack. Remember to perform this task yourself in any program that you run without a monitor.

Enabling Interrupts We mentioned that before the CPU will permit an interrupt, it checks that the requesting device has been *enabled* to perform interrupts. If the device has not been enabled the CPU will ignore the request.

Enabling devices is within the programmer's control on a standalone system. For the terminal keyboard, enabling is controlled by bit 6 of **TKS**. If bit 6 is set, interrupts are enabled; otherwise, they are not.

Bit 6 of **TKS** is clear when the computer is first switched on, but it is set by an operating system when the system is invoked. We ran under RT-11 the following program, which you should try for yourself.

```

                .TITLE      TKOFF
                TKS = 177560
START:          BIC         #100,TKS
1$:             BR          1$
                .END        START

```

This program disabled the interrupt power of the terminal keyboard. In consequence, the keyboard went “dead,” and the program could not be stopped with two ^C.

A similar program that exited instead of going into a loop did not succeed in disabling terminal keyboard interrupt. This is because the RT-11 .EXIT monitor call enables this interrupt as a precaution against just what the program tried to do.

- EXERCISES:**
- (i) Write a program to print out a message, that disables terminal keyboard interrupt during output and reenables it afterward.
 - (ii) Bit 6 of TPS enables the terminal printer to interrupt. What do you understand by a printer interrupt request? What happens when you set bit 6 of TPS? Why?

The occurrence of an interrupt does not affect bit 6 of TKS or TPS. Once one of these bits is set by a program, it remains set until that or another program clears it or until the system is stopped and restarted. Since monitors use terminal keyboard interrupt for input, the monitor program will never disable it. Consequently, we shall assume from now on that terminal keyboard interrupt is enabled.

We illustrate terminal keyboard interrupt with a routine that lets pressing any key cause an exit.

```

START:          MOV         60,R0
                MOV         #SERV,60
1$:             BR          1$
SERV:           MOV         R0,60
                .EXIT

```

The monitor does not guard against changes to the contents of device vector addresses. So our routine saves the contents of location 60 in R0 before setting location 60 to hold the address of our own interrupt service routine and going into a perpetual loop. Any keyboard interrupt will set PC to point to SERV, whereupon our routine restores the original contents of location 60, leaving intact the path to the monitor’s interrupt service routine, and exits.

One problem with our routine is that it exits while PS is still set to the contents of location 62. It so happened that any problems that this could cause were taken care of by the .EXIT call of the monitor we used. If, as will usually be the case, we want to return to our own program, rather than exit, after servicing an interrupt, we should use RTI for the return. Recall that this instruction removes PC and PS from the stack, so that the CPU is restored to its precise preinterrupt state. Of course, we can change the return address by altering (SP), just as with a TRAP instruction.

- EXERCISES:**
- (i) Amend the preceding routine so that it returns to the loop for all character inputs, except X, for which it exits.
 - (ii) What happens if the interrupt service routine goes into a perpetual loop?

We have used the interrupt facility to send a signal to stop a program but, of course, it can be used to input data. Let us construct a program to do two things “at once.” We shall write data from

a block of memory locations to the terminal printer while reading data from the terminal keyboard into another block of memory locations.

The main program saves the contents of location 60 in a register, points location 60 to the interrupt service routine **SERV**, and outputs a message whose first byte is pointed to by another register. Such an output routine was seen in Section 4.1. The program can then restore location 60 and exit.

Meanwhile, the interrupt service routine can store input data in a block of bytes pointed to by R2.

```
SERV:      MOVB      TKB,(R2)+
          RTI
```

Observe that *the interrupt service routine does not require a loop to check the ready bit in TKS*. The terminal keyboard has interrupted, so it *must* have a character available. This is a great saving in CPU time and makes I/O managed by interrupts (*interrupt-driven I/O*) considerably superior.

- EXERCISES:**
- (i) Fuse the preceding main and interrupt routines into a complete program. Draw a flowchart for it.
 - (ii) Amend the program to exit after input and output are both finished, regardless of which finishes first. Is there any other problem of lack of coordination that the program overlooks?
 - (iii) Write your own macro **.TTYIN**.
 - (iv) Write an interrupt-driven routine to read a decimal number from the terminal keyboard.
 - *(v) Write a program to sort numbers into increasing numerical order *as they are input* by going through the block where numbers are already stored in numerical order until the correct place for the current one is found and moving the rest of the block up one location. Overlap this with an interrupt-driven routine to read the numbers. Be sure that you make no assumptions about the relative speeds of different processes. Compared with the sorting program in Section 2.3, which *sorting technique* is more efficient? Which *program* is more efficient?
 - (vi) Incorporate in your program for Exercise v a routine to let you know how much CPU time (in terms of some instruction cycle as a measuring unit) is wasted by either process waiting for the other.

Terminal Printer Interrupts We have seen that having the terminal keyboard interrupt when a character is available, with the CPU free to perform other tasks meanwhile, is far superior to having the CPU waste its efforts in a loop just waiting for a character. Of course, for the gain to be realized, programs must be structured so that the CPU is given another task to perform instead of just going into some other loop to wait for the next interrupt.

In the same way, it is far preferable to have interrupt-driven output instead of keeping the CPU in a loop checking the printer status register until the printer buffer is available. The condition causing an interrupt will then be the lack of a character in **TPB** (more precisely, that the ready bit in **TPS** is set). Now it would certainly be inconvenient if lack of a character in **TPB** *always* gave rise to an interrupt; consider the effect of this on a program that performs little or no output! Clearly, we should keep the interrupt facility of the terminal printer disabled except when the program actually has data available for output.

An experimental program to display the effects of enabling terminal printer interrupt by **BIS #100,TPS** will not be effective if you run it with a monitor present. The CPU will trap through the

```

        .TITLE  BPRINT
        R1=%1
        TPS=177564
        TPB=177566
START:  MOV     64,R1
        MOV     #SERV,64
        BIS     #100,TPS
        HALT
SERV:   MOVB    #102,TPB
        BIC     #100,TPS
        MOV     R1,64
        RTI
        .END    START

```

Figure 4.3 A program to illustrate terminal printer interrupt.

vector address for the terminal printer to the monitor's interrupt service routine. This routine, after discovering that there is nothing to print out, *disables terminal printer interrupt* as part of its routine function and returns to your program.

EXERCISE: What information can you obtain by running such a program with no monitor in residence?

The vector address for the terminal printer is location 64. Let us, not for the last time, write a program to print out the letter **B** at the terminal. The program is shown in Figure 4.3. Observe that the program restores **PS** and the device vector address just as with terminal keyboard interrupt and that it leaves terminal printer interrupt disabled.

EXERCISES:

- (i) Write a program to print a message.
- (ii) Write a program to print out the contents of a given location as a decimal number.
- (iii) Write your own macro **.TTYOUT**.

If a program is to perform interrupt-driven output it is reasonable to leave the address of the interrupt service routine in location 64 throughout, restoring it when the program has completed all its output. It might seem that, as long as terminal printer interrupt is disabled when no output data are available, no harm could come of this even with a monitor in residence. Consider, however, what would happen if a key on the terminal keyboard were pressed. The CPU would respond to the keyboard interrupt by trapping through location 60 to the monitor's interrupt service routine. Part of the monitor's routine for keyboard interrupt is to echo the input character at the terminal printer. Now the monitor uses interrupt-driven output. So it puts the input character where the *printer* interrupt service routine of *the monitor* would expect to find it *and enables terminal printer interrupt*. The CPU now traps through location 64 which, however, the user has set to contain the address of *his or her* interrupt service routine. The user routine is now performed; any changes to register contents that the monitor keyboard interrupt service routine has made may adversely affect the performance of this user routine. Monitor routines will normally restore register contents (except **R0**) before returning, but this one has been interrupted before getting a chance to restore registers. When the user routine is completed, the **RTI** instruction returns control to the next address *in the monitor's keyboard interrupt service routine*. The result is wholly unpredictable.

System monitors are such complex programs that letting a user program take over part of a particular function is somewhat risky. Even the most careful study of a monitor program may overlook a dependency of some part left in use on the part that has been disabled. With regard to interrupt-driven I/O, a program that performs output should either have its own keyboard interrupt service routine or disable keyboard interrupt.

- EXERCISES:** *(i) Devise another solution to the problem just discussed by having the printer interrupt service routine check whether the user's own program caused the interrupt; if not, the user's routine should pass control to the monitor's printer interrupt service routine.
- (ii) Write a program using interrupt-driven I/O to echo characters typed in at the terminal. Your program should reserve two blocks of memory locations as data repositories, or *buffers*. Start by filling buffer 1 with input data, fill buffer 2, then go back and fill buffer 1 again, and so on. Meanwhile, as soon as buffer 1 has been filled, empty it to the terminal printer, empty buffer 2, empty buffer 1 again, and so on. You should wait until a buffer is full before beginning to empty it, and vice versa; thus, a buffer load of characters will be echoed at a time. This is an example of *double-buffered I/O*. (*Programming hint: use coroutines.*)

Priority We mentioned earlier that the CPU will only yield control of the UNIBUS to a device having sufficient priority. Thus a device whose priority level is too low will not even get the chance to send an interrupt request.

Every peripheral device with the capacity to interrupt has its own fixed priority, on a scale from 0 to 7; only priorities 4, 5, 6, and 7 are actually used. *This priority is determined by the hardware.* For example, both terminal keyboard and printer normally are set to priority 4, and a physical adjustment is needed if the level is to be changed. (On the LSI-11 the priority scale is 0 to 1 only, and all peripheral devices have priority 1.)

In addition, the CPU has its own priority on a scale from 0 to 7 (0 to 1 for the LSI-11). *The priority of the CPU is set by the program.* At any time, the CPU will yield control of the UNIBUS only to a device with priority greater than its own.

The CPU determines its own priority by looking at bits 5 to 7 of **PS**, the processor status word (on the LSI-11, at bit 7 only). The setting of these bits is the priority number. Note that these three bits do not coincide with an octal digit boundary.

Let us consider how to set the CPU to priority 7 so that no external device can interrupt. The PDP-11/45 and 11/55 computers offer users with the necessary operating privileges the Set Priority Level instruction **SPL**.

SPL 7

We use a PDP-11/45 on which this instruction, regardless of the priority level specified, inhibits *all* interrupts until the instruction following the **SPL** is completed. The manuals do not mention this phenomenon, so it is not clear if it is common to all models.

On machines without this special instruction we can perform

MOVB **#340,PS** ;priority level 7

as long as we remember that this clears the condition codes. Even **MOV** would work but, since the larger PDP-11s use the high byte of **PS**, it is better practice not to reference it indiscriminately. Note that a **BIS** instruction can also be used to set this particular priority level.

On the LSI-11 **PS** cannot be directly addressed, but there are special Move From **PS** and Move To **PS** instructions to access its low byte.

MTPS **#300** ;set priority bit

- EXERCISES:** (i) Write a program to print out the priority under which the CPU is running when it executes the program.

- (ii) Write a program that invites input of a number between 0 and 7 and sets the priority accordingly.
- (iii) Write a program that enables you to discover the interrupt priority of the terminal keyboard. (*Hint: Do not use interrupt-driven output.*)
- *(iv) Write a program that enables you to discover the interrupt priority of the terminal printer.

The astute reader will have observed that CPU priority level is automatically reset when an interrupt occurs. The preinterrupt **PS** is put onto the hardware stack and is replaced by the contents of the second word of the device vector. This word can be set under program control. Thus there are two ways to set the priority of an interrupt service routine. Suppose, for example, we want the keyboard interrupt service routine to operate with the CPU at priority 4. We then have the choice between

MOVB #200,PS

in the interrupt service routine itself or

MOVB #200,62

before an interrupt can occur. Certainly the latter form has the advantage of being performed once only, rather than every time an interrupt occurs. The important advantage of setting the priority in advance via the device vector, however, derives from the fact that, when an interrupt occurs, the hardware completes its response to it before allowing any further interrupts. Thus if location 62 has been set to contain 340, any keyboard interrupt service routine begins at priority 7 and is itself in no danger of interruption. Whereas if **PS** is reset within the interrupt service routine, even by the first instruction of that routine, it is possible for another interrupt to intervene.

It is not hard to see why it may sometimes be undesirable to allow an interrupt service routine to be interrupted. For example, a keyboard interrupt service routine may be interrupted before it can fetch the character from **TkB**; by the time the second interrupt returns, the contents of **TkB** may have decayed. It may be a good idea to keep the priority of the CPU high during an interrupt service routine until all such crucial tasks have been performed. Alternatively, if another device *must* be allowed to interrupt at any time, its interrupt service routine should be short; also, since it cannot be known in advance where the interrupt will occur, no attempt should be made to modify the return by altering the stack before **RTI**.

If these precautions are taken, there is no problem at all in *nesting* interrupts in the same fashion as subroutines and traps. Indeed, any mixture of subroutines, traps, and interrupts may be nested, with proper use of **RTS** and **RTI** ensuring that the correct returns are made.

EXERCISES: (i) Can a device's interrupt service routine be interrupted by the same device?
 (ii) Rework all your interrupt-driven I/O programs to incorporate proper considerations of priority.

If the CPU receives simultaneous interrupt requests from devices with different priorities, it will first grant the higher-priority request. Between devices of the same priority, proximity to the CPU on the UNIBUS is determinative. This, of course, refers to the priority of the device itself as determined by the hardware; it does not mean the priority of the CPU while it is running the device's interrupt service routine as set by the program.

The device that failed to interrupt will not be forgotten. Its interrupt request will be serviced as soon as CPU priority allows; meanwhile, the interrupt is *pending*. Usually, the interrupt service routine for the high-priority device will be performed with the CPU at high priority; the lower-priority

device's request will then remain pending until the other device's service routine has run its course and returned.

One of the highest priority devices that a system is likely to possess is a clock. The clock "ticks" by requesting an interrupt. The frequency of ticks is that of the electrical supply: sixty times per second in the United States, fifty times per second in most other countries. The clock uses locations 100 and 102 for the vector to its interrupt service routine. Location 177546 is the clock status register, with bit 6 enabling interrupts as usual.

An interrupt service routine for the clock can be used to keep track of the time by counting ticks. Note that if the clock's interrupt request is not granted within 1/60 second, the next tick will have no way of distinguishing itself; the CPU can only detect the presence or absence of an interrupt request. Thus a program that counts ticks must be careful that the CPU never operates at a priority greater than or equal to that of the clock for long enough to lose a tick.

- EXERCISES:**
- (i) Why does the clock have no data register?
 - (ii) Write a program to ring the bell (^G) at one-second intervals. Have you made any assumptions about how long it takes to execute your clock interrupt service routine? Are they warranted? (You may find the WAIT instruction helpful; it stops the CPU, with PC pointing to the next instruction, until an interrupt occurs.)
 - (iii) Write a program to enable you to determine the priority of the clock.
 - (iv) Write a routine that, when incorporated into a program, prints out when the program exits the CPU time that the program has taken.

T Bit It is impossible to understand fully the interplay between trace traps and interrupts without a detailed knowledge of the hardware. The topic is confusing, and we can do no more than point out a few pitfalls.

We shall assume that the T bit has been set by setting bit 4 in 2(SP), followed by RTI. Indeed, on most models of PDP-11 CPU, the T bit can only be set indirectly by an instruction such as RTI or TRAP; an attempt such as

BIS #20,PS

simply will not work, and neither will use of the console D switch.

The normal sequel is that the CPU performs one instruction following the one that set the T bit and then traps. Thus, if RTI sets the T bit, one instruction in the routine to which RTI returns control will be performed before the trace trap occurs. Note that if the instruction to be performed is SPL, the trap is delayed until the CPU has also performed the instruction following the SPL.

Problems can occur if the routine (presumably ODT) that set the T bit was running with the CPU at high priority, but the one to which RTI returns control is at low priority. Suppose that an interrupt was pending when the RTI was performed and that the interrupt has a higher priority than the program now in control. Before fetching the instruction to be single stepped, the CPU responds to the interrupt and loads the new PS and PC. Thus the T bit is no longer set, and the interrupt service routine runs its course without causing a trace trap, returning with its own RTI. This RTI, for technical reasons, causes the CPU to "notice" immediately that the T bit is set. Consequently, it will trap *before* performing the instruction to be single stepped.

This can be cured by having the interrupt service routine return with the instruction RTT* instead of RTI. The only difference is that RTT delays the occurrence of the trace trap for one more instruction than RTI. Thus the desired instruction would be performed, whereupon the CPU "notices" the T bit set by RTT and traps.

*Not available on some smaller CPUs.

Suppose, however, that no interrupt was pending, but that one occurs after the CPU has fetched the instruction to be single stepped. The CPU will complete that instruction, perform the interrupt service routine, and return. Now, however, a return using RTT would inhibit the trace trap until yet one more instruction has been performed, so that a single-step command to ODT would result in the performance of *two* program instructions. In this case the proper return would be with RTI.

Unfortunately, the very nature of an interrupt makes it possible that either of these cases might occur, so there would seem to be no correct way to effect the return from the interrupt service routine. This results in anomalies when debugging programs with interrupt-driven routines.

4.3 STORAGE DEVICES: MONITOR CONTROL

Until now, any information required by our programs has been typed in at the terminal at execution time, and the results of computations have been typed out at the terminal. In this and the next sections we consider how a program may obtain data from, and pass data to, a file on a storage device. The general approach is the same, regardless of whether the storage device is any of the various forms of disk or tape. The focus in this section is on monitor-controlled I/O; an introduction to the complexities of user-controlled I/O appears in Section 4.4.

Monitor Calls We shall describe monitor-controlled I/O in terms of the RT-11 system macros. As in Section 1.4, you should determine whether these are available on your system and, if they are not, what replaces them.

The RT-11 operating system has developed through several versions; the most recent at the time of writing is version 3B. Most of the system macros introduced in this section (but not those discussed earlier) differ, in calling sequence and code generated, according to the version of RT-11 being used. We shall study only the latest version of these macros. If your system has version 3 or 3B of the RT-11 system macros library, you can use these macros without emendation, as long as you issue an **.MCALL** directive for each of them to tell the assembler to search the system macros library, **SYSMAC.SML**. A typeout of that file will include, near the beginning, a statement of its version number.

As we observed in passing in Section 1.4, the assembler and the system macros library must be compatible. We use one PDP-11 system that possesses all versions of the RT-11 MACRO-11 assembler but only version 2 of **SYSMAC.SML**. Unfortunately, version 2 library files are in a format that a version 3 assembler cannot read, so to use system macros we must either write our own calling sequences to the monitor's **EMT** routines or use a version 2 assembler.

The system macros to be discussed will not work correctly if submitted without modification to an earlier version of the assembler. Differences in the ways in which parameters are passed will even result in assembly errors; version 3 calls pass the address where a number intended for the low byte of an **EMT** instruction is housed, but version 2 assemblers expect the actual number to be passed, so they try to encode an **EMT** with an address in its low byte. (Why does this generate an *assembly error*?)

You can use a version 3 macro call with a version 2 assembler and library, as long as you include in your program a call on the system macro **.V2.** to perform the translation to version 2 format. Note that the name of this macro contains *six* characters. Thus your program should include

```

      .MCALL      .V2. .
      ...
      ...
START:  .V2..

```

after which version 3 calls will be correctly interpreted by a version 2 system. If your system is version 1, replace `.V2.` with the `.V1.` system macro. (If you put both macros in your program to make it work on any system, it will work on none.)

EXERCISE: Find out how `.V2.` works.

Radix-50 Code A great convenience of having the monitor organize communication between program and storage device on our behalf is that we can use files referred to by name in the way that is already familiar to us. A program that manages this I/O itself, however, must handle the troublesome details of selecting and locating the actual physical spot where its data reside on the device.

A program must therefore be able to pass file names to the monitor. On the PDP-11, systems programs encode such names not in ASCII code but in *radix-50 code*. This is a code in which only uppercase letters, numerals, `.` (period), space, and `$` may be represented. The codes are:

<i>Symbol</i>	<i>Code</i>
Space	0
A	1
.	.
.	.
Z	32
\$	33
Period	34
(Unused)	(35)
0	36
1	37
2	40
.	.
.	.
9	47

Thus, the symbols used in file names are encoded into the set of numbers less than (octal) 50.

We may regard these symbols as numerals in a number system in which the base, or radix, is (octal) 50. In such a system, XY would be interpreted as: X in the "50s" column, Y in the "units" column. Since X, in this code, represents 30 and Y represents 31, XY would represent the number $(30 \times 50) + 31 = 1731$ (octal calculation). Similarly, YX would represent $(31 \times 50) + 30 = 1750 + 30 = 2000$.

Likewise, the three-"digit" radix-50 number XYZ would represent the octal number

$$(30 \times 50 \times 50) + (31 \times 50) + 32 = 113000 + 1750 + 32 \\ = 115002$$

Observe now that this representation of the three symbols XYZ encodes into a single PDP-11 word. Indeed, the largest three-"digit" radix-50 number is 999 which, in octal notation, is

$$(47 \times 50 \times 50) + (47 \times 50) + 47 = 171700 + 3030 + 47 \\ = 174777$$

This also encodes into a PDP-11 word. Radix-50 code therefore allows us to put three characters into a single 16-bit PDP-11 word, as long as the characters are taken from the special radix-50 set.

EXERCISE: Write a program to accept input of three characters and print out their radix-50 encoding.

The **.RAD50** directive is used to assemble a string of characters packed three to a word in radix-50 coding. The syntax of this directive is the same as that of **.ASCII**. If only one or two characters remain for the last word, they will be left-justified in the word, as if spaces had been supplied to fill it (remember that space encodes as 0). Observe the coding, with **#** representing typing a space.

```
114750      .RAD50      /XY/      ;like /XY#/
001731      .RAD50      /#XY/
```

Device Handlers For each device with which the monitor is able to communicate, there is a special program, part of the operating system, called a *device handler*. The handler for such a frequently used device as the disk will probably be kept in memory as part of the resident monitor. Other devices may then have their handlers stored on the disk, to be brought into memory when needed.

Before a device can be accessed, its handler must be brought into memory. If this has not already been done by the operating system, the program must do it by issuing the **.FETCH** monitor call. Unless you know your operating system very well, you should always issue a **.FETCH**; no harm is done if the device handler is already resident in memory.

The **.FETCH** monitor call takes two arguments, separated by commas: the address where the device handler is to be loaded, and the address where the *device name* is stored. Thus

```
.FETCH      #HNDLER,#DEVNAM
```

causes the monitor to check location **DEVNAM** for the device name and load the appropriate handler into memory locations starting at location **HNDLER**. Observe the syntax of this call, with immediate mode for both arguments.

At location **DEVNAM** there must appear a two-letter radix-50 code giving the device name. Examples of codes are: **RP04** disk, **DK**; **RK06** disk, **DM**; **DECtape**; **DT**; cassette tape, **CT**; punched card reader, **CR**; and paper tape, **PC**. Taking the **RP04** disk as our example, we would have

```
DEVNAM:      .RAD50      /DK/
```

We should make no assumptions as to how much space the handler will take up. A reasonable approach is to reserve location **HNDLER** immediately before the **.END** statement, so the device handler can extend into free memory. The monitor will complete its response to **.FETCH** by setting **R0** to point to the first location following the end of the device handler. If the handler was already in memory, the monitor will just point **R0** to location **HNDLER** and will not load a further copy of the handler.

The situation so far is

```

      .MCALL      .FETCH
START:  ...
      ...
      .FETCH      #HNDLER,#DEVNAM
      ...
DEVNAM:  .RAD50      /DK/
      ...
      ...
HNDLER:  ...
      .END      START
```

Note that this does not actually reserve space for the handler. (How could we do so? Why might we need to?)

We can load handlers for a series of devices by

```
.FETCH    #HNDLER,#DEV1
.FETCH    R0,#DEV2
.FETCH    R0,#DEV3
```

and so on. Again, note the syntax: in the first call, the address to be passed is that of location **HNDLER**, so immediate mode is required; thereafter, the *contents* of **R0** give the address at which to start loading the handler.

RT-11 system macros will clear the C bit on successful performance of their function and set it if an error occurred. I/O operations to storage devices can fail for a variety of reasons, many of which (such as a disk drive being switched off) are outside the control of a program. Thus it is essential to follow each monitor call with a **BCS** to a routine that prints out a helpful error message and either exits or accepts instructions from the terminal as to what to do next.

EXERCISE: What would you deduce about a computer system in which the disk handler was not resident in memory?

Opening a File Let us write a program to open a file to be called **IOTEST.DAT** on the disk. We shall have to include in our program a four-word block specifying the file name, as follows:

```
FILNAM:    .RAD50    /DK/
           .RAD50    /IOTESTDAT/
```

Of course, the single statement

```
FILNAM:    .RAD50    /DK#IOTESTDAT/
```

(where **#** indicates typing a space) does the same thing, but it is less clear. The monitor call will take the device name from the first word of this block, the file name from the second and third words, and the file name extension from the last word. Thus, if the desired file were **IO.DAT**, we would have **/IO####DAT/** in the **.RAD50** statement, with four spaces typed to fill out the code. Note that the period introducing the file name extension will be supplied by the operating system and should not be included in the **.RAD50** statement.

The first word of the **FILNAM** block may, as before, be used for referencing by the **.FETCH** monitor call. The block has, however, been set up with the **.ENTER** monitor call in mind. This call is used to reserve space on the specified storage device under the specified file name.

Once a file has been created with the **.ENTER** call, references to it will be made, not via the block at **FILNAM**, but simply by giving a number associated with the file. This number is assigned by the **.ENTER** call and specifies the *channel* on which the file has been opened. Under RT-11 the channel number may be in the range **0** to **377** (octal). A channel is not a physical connection between the computer and the storage device; it is merely an identifying number for a particular set of I/O operations.

The **.ENTER** monitor call takes five arguments, separated by commas. Considering them out of order, the fifth argument is used only for certain tape operations; we shall leave it blank. The second argument is the chosen channel number; we may as well use the first channel, number **0**, giving us so far

```
.ENTER    ?,#0,?,?
```

where the ? remain to be filled in. Note that the value 0 for the channel number must be passed as a proper assembler language argument. We could also have used

```
CLR      R1
.ENTER   ?,R1,?,?
```

We could not, however, have used R0 in this way, because .ENTER begins by pointing R0 to a four-word block that it sets up for reference by the EMT routine that the macro then calls. This four-word block (in addition to the one at FILNAM) must be reserved by the program. The same EMT reference block can be used by all of the program's I/O calls. Some of these require more than four words; in order to leave plenty of room for these and any calls that future monitor versions may make available, it is customary to have

```
IOBLK:    .BLKW      10
```

reserving eight words.

The first argument in the .ENTER call must be the address of the first word of this EMT argument block. The .ENTER macro will set up this block as an argument list and then issue an EMT instruction.

The third argument in the .ENTER call must be the address of the first word of the block specifying the file. So far we have

```
                .MCALL      .ENTER
START:          ...
                ...
                .ENTER      #IOBLK,#0,#FILNAM,?
                ...
FILNAM:         .RAD50      /DK/
                .RAD50      /IOTESTDAT/
                ...
IOBLK:          .BLKW      10
```

The final argument for us to consider is the fourth. This must specify the amount of space to be reserved on the storage device for the file to be created. If -1 is passed as this argument, the largest piece of empty storage space on the device will be reserved. Doing this, our monitor call to open the disk file IOTEST.DAT is

```
.ENTER      #IOBLK,#0,#FILNAM,#-1
```

This should be followed by a BCS to a routine to print out a message such as ENTER FAILED and then proceed according to the ingenuity of the programmer.

Although disk space has now been reserved for IOTEST.DAT, the monitor does not yet regard it as a permanent file; in particular, no entry has been made for it in the user's directory. The entry will be made permanent when the monitor call .CLOSE for the specified channel is issued.

```
.CLOSE      #0
```

This call, exceptionally, does not set the C bit on an error; instead, the monitor will intervene. Therefore this call should not be followed by a BCS. Like all these I/O system macros, it uses R0, whose contents may therefore be changed.

- EXERCISES:**
- (i) Write the complete program to add IOTEST.DAT to your directory of disk files. How much disk space does the file occupy? What is in it?
 - (ii) Use the editor to write something into IOTEST.DAT. Now run your program again and check what happens.
 - (iii) Research your system's facilities for *protecting* files. Protect IOTEST.DAT to "read only" status; then run your program again.

Writing a File Instead of just obtaining a directory entry for an empty file, let us interpose between .ENTER and .CLOSE a command to write something in the file. We shall, inevitably, write a program to create a file containing the ASCII text B.

To write into the file already created with .ENTER, we use the .WRITW macro call. This takes five arguments, of which the first two are just as in the .ENTER call: the address of the EMT argument block, and the channel number that an .ENTER call for the file has already specified.

```
.WRITW      #IOBLK,#0,?,?,?
```

The third argument passes the address in memory at which data destined for the storage device are to be found. If we declare elsewhere in the program

```
MEM:        .WORD      102
```

we can call

```
.WRITW      #IOBLK,#0,#MEM,?,?
```

The fourth argument passes the number of words to be written by the .WRITW call; in this case the number 1 must be passed.

The fifth argument specifies which *block* of the file is to be written. Storage space on disks and magnetic tapes is accessible not in individual words but only in blocks of words. The size of the block is a characteristic of the storage device within the system: for the disk it is (decimal) 256 words. We wish to write into the first (and only) block of our file, block 0. Thus the call is

```
.WRITW      #IOBLK,#0,#MEM,#1,#0
```

- EXERCISES:**
- (i) Complete the program.
 - (ii) Amend the program so that it writes C instead of B. Run it again. What has happened to the original IOTEST.DAT?

Note that the monitor TYPE command cannot, in most systems, be used to type out IOTEST.DAT, since TYPE ignores text not terminated by a ↵. An editor must be used to check the contents of the file.

Suppose now that we want to output the whole alphabet to IOTEST.DAT. We can do this one word at a time; starting with MEM containing the number 101 and R2 clear, we would perform

```

      MOV      #32,R1
LOOP: .WRITW    #IOBLK,#0,#MEM,#1,R2
      BCS      WERR
      INC      R2
      INC      MEM
      SOB      R1,LOOP
```

Here the block number is passed in the `.WRITW` call as the contents of `R2`. If we neglected to increment `R2` in the loop, each `.WRITW` would output to the first block of `IOTEST.DAT`, starting its write operation at the beginning of the block *each time*. The result would be a one-block file, containing just the ASCII code for `Z`.

This routine does not utilize the disk in the most efficient way, since the file it creates is twenty-six blocks long (check your directory), with one letter to a block. Thus we use over six thousand words of disk storage to house twenty-six bytes of information. Instead, we should have the alphabet set out in our program as twenty-six bytes of ASCII code starting at `MEM`, and

```
.WRITW      #IOBLK,#0,#MEM,#13.,#0
```

Note that the assembler accepts a period after a number as indicating decimal notation, so that `13. = ^D13 = 15`.

Repeat Directives An elegant way to have the assembler create the block at `MEM` is to use the *assembler macro directive* `.IRPC` to Indefinitely RePeat with Character substitution. The form of the call is

```
.IRPC      X,ABCDEFGHIJKLMNOPQRSTUVWXYZ
.ASCII    /X/
.ENDM
```

Here `X` is a dummy argument, to be replaced successively by the characters following the comma in the `.IRPC` directive. This assembler macro would generate the code

```
.ASCII    /A/
.ASCII    /B/
```

and so on, through to

```
.ASCII    /Z/
```

The expansion of such a macro would make a listing program clumsily long. A program that uses the `.LIST ME` directive to list macro expansions can turn this facility off with `.NLIST ME`. It may be convenient to precede an `.IRPC` with this directive and follow it with a fresh `.LIST ME` to resume listing of macro expansions.

Observe also the following use of `.IRPC` to generate code to put the contents of `R0` to `R5` onto the system stack.

```
.IRPC      X,012345
MOV        R'X,-(SP)
.ENDM
```

The `'` symbol serves in MACRO-11 to delimit an argument; without it, this code would contain the undefined symbol `RX`. The assembler will remove the `'` when it expands the macro and, in this case, generate

```
MOV        R0,-(SP)
MOV        R1,-(SP)
```

and so on, through to

```
MOV      R5, -(SP)
```

Here we could also use the Indefinite RePeat directive `.IRP`. The call would be

```
.IRP      X,<R0,R1,R2,R3,R4,R5>
MOV      X, -(SP)
.ENDM
```

to generate exactly the same code. Although `.IRPC` is tidier in the present case, `.IRP` is more generally applicable when argument substitution is required.

I/O Buffering In order not to waste space, a program should output data to a storage device in packages equal in size to a block on the device. Thus, if we write (decimal) 256 words in one `.WRITW` to the disk, we shall exactly fill a disk block. Note that there is no way of getting a `.WRITW` to start storing its data anywhere but at the beginning of a disk block.

We need, therefore, to maintain a *buffer* of (octal) 400 words. When the buffer is filled, it will be output to the disk. We shall reserve a word `OBUF` to hold the address of the first word of our output buffer. Suppose that we want to store on disk the successive results that, say, subroutine `CALC` returns in `R0`. The following sequence will fill our buffer with data and output it to the disk.

```

                                MOV      OBUF,R1                ;point R1 to buffer
                                MOV      #400,R2                ;word count in R2
LOOP:                          JSR      PC,CALC                ;do our sums
                                MOV      R0,(R1) +              ;CALC returns in R0
                                SOB      R2,LOOP
                                .WRITW   #IOBLK,#0,OBUF,#400,#0
```

The buffer is said to be *emptied* to the disk although, of course, it still contains our data.

Observe that the buffer address is passed as the contents of location `OBUF`. If we want to write more than one block, we must pass the block number in a similar fashion.

- EXERCISES:**
- (i) Write a program to create on the disk a file containing, in successive words, the octal numbers:
 - (a) 1 to 2000.
 - (b) 0 to 2000.
 - (c) m to n , where m and n are input at execution time.
 - (ii) Write a program that accepts text typed in at the terminal and terminated by \$ (ESCAPE) and writes it into a disk file.
 - *(iii) Amend your program for Exercise ii so that you can input at the terminal the name of the file to be created.
 - (iv) Write a program that creates two disk files and accepts text typed in at the terminal, sending it to the first file until ^B is typed. Text should now go to the second file until ^A is typed, and so on. (Each file must have its own channel.)
 - (v) Can a single file be open on two channels simultaneously?

A single `.WRITW` command can, in fact, be used with more than a single block full of data. As many successive blocks as are needed will be filled on the disk. A program that writes, say, two disk blocks at a time must be careful to increment the block number in the `.WRITW` call by 2 between writes.

Reading a File The call to access an already existing file is `.LOOKUP`. The appropriate device handler must be in memory when this call is issued. The argument list for `.LOOKUP` is similar to that for `.ENTER`, except that there is no argument to specify file length. Thus we can access a file on channel number 0 by

```
.LOOKUP    #IOBLK,#0,#FILNAM
```

where the block at `FILNAM` specifies the desired file and that at `IOBLK` is for use by the system macro, just as before.

Having accessed the file, we can use the `.READW` monitor call to bring specified blocks into memory. The form and usage of the `.READW` call are the same as for `.WRITW`, except that data are transferred in the opposite direction, from a disk block to a memory buffer; this is called *filling* the buffer from the disk.

It is also possible to update an existing file by issuing `.WRITW` calls after `.LOOKUP`. On some systems this will work properly only if the file was originally written with full blocks, as just described.

When reading a file, we must be able to determine when we have reached its end. An attempt to read past the end of a file will set the carry bit but, since hardware problems can also do so, this does not offer a conclusive check. However, `.LOOKUP` returns with `R0` containing the number of blocks in the file accessed by the call. The program must use this information to ensure that no attempt is made to read beyond the end of the file.

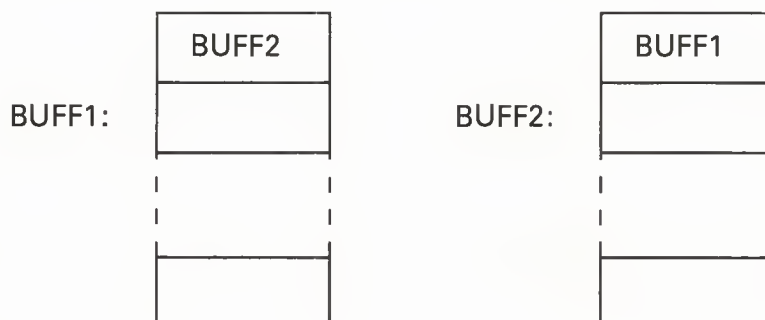
- EXERCISES:**
- (i) Write a program to print out, in a pleasing format, the file containing the octal numbers 1 to 2000 that you created in Exercise i on p. 134.
 - (ii) Write a program to replace, in this file, the numbers 1001 to 1400 with 3001 to 3400, leaving all other locations in the file unchanged.

Multiple Buffering The `.READW` and `.WRITW` calls return control to the user program only after their operation is fully completed. This means, as discussed in Section 4.2, that time is wasted waiting for I/O to finish.

A program can instead use the `.READ` and `.WRITE` monitor calls (same syntax). These calls not only set up the locations that control I/O on the storage device but also set the device's interrupt enable bit and immediately return control to the user program. The sequel is familiar: the program can go on to do something else, and the device will interrupt when it is ready.

Consider once again our routine to fill a buffer with the results of calculations and then empty the buffer to the disk. Instead of waiting while this output is completed, the program can proceed with the next batch of calculations. It must not, however, interfere with the buffer currently being emptied until that I/O operation is completed. Another buffer must be provided for the program to deposit its data while the first buffer is being emptied. When this buffer is full, a `.WRITE` can be issued to empty it, and the program can again turn its attention to the first buffer.

A nice way to achieve smooth transfer between buffers is to precede each buffer with a *header* word that contains a pointer to the first word of the other buffer.



We maintain R1 as a pointer to the first word of the current buffer, beginning with `MOV #BUFF1,R1`. This time R1 will not be changed by the loop that performs calculations and fills a buffer so, using R3 as a block pointer, we have

```

LOOP:      perform calculations
           and fill a buffer
           .WRITE      #IOBLK,#0,R1,#400,R3      ;empty the buffer
           INC          R3                        ;next block
           MOV          -2(R1),R1                 ;change buffers
           BR           LOOP                       ;go on calculating

```

Note that the routine to perform calculations and fill a buffer with their results does not even “know” which buffer it is working on at any time, in the true spirit of structured programming.

We can use the same method of changing buffers with any number of buffers. The header word for each buffer will point to the first word of the next buffer; the header for the last buffer points back to the first. Such a configuration is called a buffer *ring*. There is, however, not often much benefit in having more than two buffers.

The program must take care that the calculating routine does not get so far ahead of I/O operations that it tries to fill a buffer before its previous contents have been emptied to the storage device; calculation must not be permitted to pass I/O. (Is there a converse problem?) Unfortunately, `.READ` and `.WRITE` provide no indication of when the data transfer is completed. The `.WAIT` monitor call may be issued to bring the two program processes into step. This takes a channel number as its only argument and suspends program execution until all pending I/O operations on that channel have been completed.

Alternatively, a program can use the `.READC` and `.WRITC` monitor calls. Each of these calls takes six arguments: the first four as previously; the address of the user's *completion routine* as the fifth argument; and finally, the block number. As with `.READ` and `.WRITE`, the monitor will return control immediately to the user program. When the I/O operation is completed, however, the monitor will respond to the device interrupt by putting the current user program PC onto the stack and transferring control to the completion routine specified in the `.READC` or `.WRITC` call.

- EXERCISES:**
- (i) Write a program using double buffering to create a disk file containing, in successive words, the first two thousand prime numbers.
 - (ii) Write a program to replace the number in each word of a given file with its smallest prime factor. (*Hint.* Access the file created in Exercise i.)
 - (iii) How should a completion routine return to the main program?

4.4 STORAGE DEVICES: USER CONTROL

The purpose of this section is merely to provide an overview of the process whereby a storage device is controlled. It is beyond the scope of this book to offer more than a general introduction to this highly complex subject.

The model for our discussion is the RK06 disk system. This is a popular modern storage device, with quite typical control characteristics. Armed with the appropriate user's manual, you can easily interpret our description of the RK06 in terms of any other storage device that your system may possess.

The RK06 disk system consists of a cabinet equipped with control devices (the RK611 *drive controller*) and housing a *disk cartridge*. A system may possess many cartridges, and the computer user can put into the cabinet whichever cartridge is required for the task in hand. When its protective casing is removed, the cartridge has the appearance of two phonograph records (called *platters*) stacked about

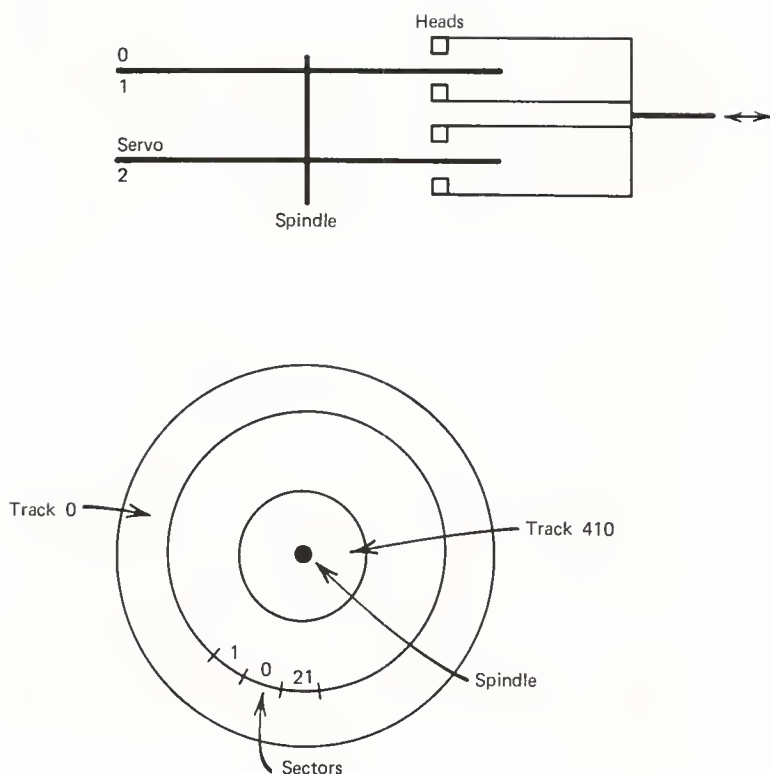


Figure 4.4 An RK06 disk configuration.

two inches apart on a spindle. The platter surfaces, however, are magnetic storage media rather than wax; during data transfer operations, they rotate at 2400 revolutions per minute, not $33\frac{1}{3}$.

The drive controller is equipped with four retractable rods that project over the platter surfaces. At the end of each rod is a *head*, somewhat akin to that in a tape recorder. A diagram of the platter and head configuration appears in the upper part of Figure 4.4. Note that all four heads are extended or retracted over the platters simultaneously.

One of the four platter surfaces is the “servo” surface, prerecorded with timing and position information for its head to pick up; the head for this surface is *read only*. The other three surfaces are available for user storage, and they have *read/write heads*.

The surface of each platter is divided into (decimal) 411 concentric rings, or *tracks*, with track 0 the outermost. This is diagrammed in the lower part of Figure 4.4. In spite of the differences in track lengths, each track offers the same amount of data storage: (decimal) 107520 bits.

Referencing a particular track on a platter is a two-stage process. Suppose, for example, that we want track 200 on surface 1. First, we must instruct the controller to move the heads to track 200. Since, however, all four heads move together, this command specifies only the “cylinder” at the radial distance from the spindle common to the tracks numbered 200 on all the surfaces. Thus, the second step, completing our specification of the track desired, is to tell the controller that it is on surface number 1.

Device Registers Instructions are issued to the drive controller by loading the UNIBUS locations used by the storage device. These locations are called the *device registers*. Most of these locations cannot be loaded by setting the console switches and using the D switch; they require a MOV, BIS, or BIC from the CPU. The locations used by the RK06 disk are referenced by a program as locations 177440 to 177476. The need for sixteen registers gives some warning of the intricacies of programming this device.

Let us consider how to move the heads to cylinder 200. If the system has more than one disk drive cabinet, we must first decide which drive we want to control. A system can operate up to eight of these drives, and each will have its number (from 0 to 7) marked on the "ready" button on the cabinet. We shall suppose that drive 1 is to be accessed. We recommend that you press the "write protect" button on the cabinet of *every* drive whose cartridge contains anything useful until you gain expertise in controlling the device.

A disk drive is selected by setting bits 0 to 2 in the RK06 *control and status register number 2*, program address 177450, mnemonic RKCS2. However, before writing into a disk register, we must check that the controller is ready to receive commands. The controller indicates its readiness by setting bit 7 in the *control and status register number 1*, RKCS1 = 177440. Thus, to select drive 1:

```

1$:      MOV      #RKCS1,R1
         TSTB     (R1)
         BPL      1$
         MOV      #1,RKCS2

```

Note that *the contents of device registers can be changed only by word instructions* and not by byte instructions.

The next step is to send the controller the "pack acknowledge" message, indicating that the CPU wishes to transmit instructions. Messages and instructions are sent to the controller by setting the bits representing the appropriate function code in RKCS1. Pack acknowledge has function code 3. The controller will respond by setting the controller ready bit (sending pack acknowledge will have cleared it); the program cannot proceed with disk control operations until this response is received.

```

2$:      MOV      #3,(R1)                ;R1 still points to RKCS1
         TSTB     (R1)
         BPL      2$

```

Having been assured that the *controller* is ready, the program must go on to check that the *drive* is ready. The one does not imply the other; the controller may be ready while, for example, the drive heads are still positioning themselves in response to the previous command. For the state of the drive we must refer to the *drive status register*, RKDS = 177452. Four bits in this register must have been set (by the device: this is a read only register) before we can proceed. Bit 15 is set to indicate that the register contains current information for the selected drive; an information update is put into RKDS in response to the pack acknowledge message, so this bit should, indeed, be set. Bits 0 and 7 are set to indicate, respectively, that the drive is available to the controller and ready to receive commands. Bit 6 is set in response to pack acknowledge to indicate that various technical matters are in order. Later commands will not work until these bits are *all* set, so the program must again wait.

```

         MOV      RKDS,R2
         COM      R2
         BIT      #100301,R2
         BNE      1$                    ;try again from beginning

```

It is necessary to complement the bits before the BIT instruction; an immediate BIT #100301,RKDS followed by BEQ 1\$ would assume that all was well even if only one of the four bits were set. Certain error conditions can clear the drive select bits in RKCS2 and, for technical reasons, a fresh pack acknowledge may be needed before we check the drive again. Hence we make our return to 1\$, the beginning of our drive control routine.

The entire procedure so far must be implemented for every new command to the disk. Let us assume that it has been written as subroutine CHKDSK, using PC as linkage register.

When **CHKDSK** returns, we can issue an instruction to the disk controller, knowing that nothing hinders its implementation (subject to our discussion on error conditions at the end of this section). To position the heads at cylinder 200, we first set up the *desired cylinder register*, **RKDC = 177460**.

```
MOV      #200.,RKDC
```

The heads are moved by setting the function code 17 for the “seek” command in **RKCS1**.

```
MOV      #17,RKCS1
```

More precisely, the seek command is encoded as 16. Bit 0 of **RKCS1** is the *go bit*, which must be set in order to execute the command encoded in bits 1 to 4. Thus the instruction is “seek and go.” Once the disk controller has accepted this command, its own circuitry takes over and the program has no more to do.

Formatting Each track on a disk platter is divided into individually accessible units of data storage called *sectors*. This is illustrated in the lower diagram of Figure 4.4. Unlike the division of each platter surface into 411 tracks, the division of each track into sectors is not a physical characteristic of the platter. Rather, it is the result of *formatting* the surface, which is achieved by running a special program. Some storage disks (such as floppy disks) are sold already formatted, but the **RK06** must be formatted by the user. The process of formatting an **RK06** disk is too complex for discussion here (the DEC operating system program that does the job is over six thousand lines long), so we must assume, as is likely, that the disks in your system have been formatted. It is not feasible to read or write a nonformatted disk.

On a formatted disk, each track is divided into 22 sectors. Each sector contains three words of *header* and 256 words (sixteen bit) of data. (An alternative method of formatting allows eighteen-bit data words, in which case there is room for only 20 sectors per track). There are gaps between header and data and between sectors to enable the drive mechanism to distinguish the different regions.

The header for each sector contains a record of its cylinder, surface, and sector numbers. It contains nothing approximating a file name; an operating system program is needed if disk addresses are to be associated with file names. If there is no such program, we must keep our own record of where things are on our disk.

Writing and Reading the Disk Suppose, for example, that we want to write sector 13 on track 100 of surface 1. Let us fill the 256 words of this sector with the successive numbers 1 to 400 (octal). Our program must reserve a buffer in memory, starting, say, at **OBUF**, to house the data to be output, and load word locations **OBUF** to **OBUF + 776** with the successive numbers 1 to 400.

Data transfers between memory and disk can take place only while the disk is rotating. Some disks are designed to rotate continuously; the **RK06**, however, may have its spindle stopped for a variety of reasons. We must, therefore, begin by issuing the “start spindle” command, encoded as 10 in **RKCS1**.

```
JSR      PC,CHKDSK      ;return when disk ready
MOV      #11,RKCS1      ;start spindle & go
```

Unless the spindle is already rotating, the next call to **CHKDSK** may loop many times within the routine until the drive ready bit is set, since the spindle must reach full speed before the controller will set that bit.

Next, the device registers must be set up for the write command. However, the relevant registers must not be changed while the go bit is still set. Thus, we have the following choices.

1. Keep testing the go bit until it is cleared by the controller.
2. Keep testing the controller ready bit until it is set by the controller; the controller clears this bit when a program sets the go bit, and the controller sets it when it clears the go bit.
3. Call subroutine **CHKDSK**, which incorporates the second choice.

For simplicity, we shall adopt the third choice. On the return, both controller and drive must be ready, so we can proceed both to set up the registers and to issue the write command. We must set **RKDC** to indicate cylinder 100.

```
MOV      #100.,RKDC
```

The desired surface and sector are specified in the *disk address register*, **RKDA**=177446. The surface number (here 1) goes into the high byte, the sector number (here 13) into the low byte. Remember, however, that byte moves to **RKDA** will not work.

The *bus address register*, **RKBA**=177444, must be loaded with the memory address from which data transfers are to start. In this connection, you should observe the comments in the note on address relocation later in this section.

The *word count register*, **RKWC**=177442, must be loaded to reflect the number of words to be transferred. This is done by loading the *twos complement* of that number into **RKWC**. In the present case we are transmitting 400 words, so we set **RKWC** to -400.

Finally, we can issue the "write" command, encoded as 22 in **RKCS1**, remembering to set the go bit at the same time. The entire routine, after starting the spindle, is:

```
JSR      PC,CHKDSK           ;let spindle reach full speed
MOV      #100.,RKDC          ;cylinder 100
MOV      #1,R0               ;surface 1
SWAB     R0                  ;in high byte
ADD      #13.,R0             ;sector 13
MOV      R0,RKDA
MOV      #OBUF,RKBA          ;output buffer
MOV      #-400,RKWC          ;word count
MOV      #23,RKCS1           ;write and go
```

Note that we have not issued a seek command before writing to the disk. The write command, in fact, incorporates a seek to the required cylinder. When the seek is completed, the head for the selected surface starts a "search" on the chosen track (cylinder 100 is, when restricted to surface 1, just track 100 on that surface) for the specified sector. When, by checking headers, the sector is found, the data transfer begins. Starting from the address in **RKBA**, words are transferred from memory to the sector, beginning always, for each write command, at the first word of the sector. This entire sequence of operations is performed by the controller in response to the write command without further program intervention.

After each word transfer, the controller increments **RKWC** by 1 to maintain a count of the words remaining to be transferred and increments **RKBA** by 2 to point to the next memory location from which data are to be transferred. As soon as **RKWC** reaches zero, the controller stops transferring data from memory. If this happens when the current sector is not yet completely written, *the rest of the sector is written with zeros*, without further change to **RKWC** or **RKBA**.

Each time, during a write operation, that a complete sector is filled, the controller increments the low byte of **RKDA** to point to the next sector. If, however, the low byte of **RKDA** is already set to (decimal) 21, the controller will clear the byte and increment the high byte of **RKDA** to point to the first sector of the *same track* on the *next surface*.

If **RKDA** already points to sector 21 on surface 2, the controller will clear **RKDA** and increment

RKDC. The purpose of having sectors written in this order is to perform the time-consuming operation of moving the heads as rarely as possible.

After moving on to the next sector in this fashion, the controller checks the word count in **RKWC**. If data remain to be transferred, the controller will write the sector now pointed to and repeat the process.

When, at the end of a sector, **RKWC** is eventually found to be zero, the controller sets the controller ready bit and stops operations. Note that **RKDA** and **RKDC** together now point to the sector that, under this scheme of ordering, follows the last one written.

Reading the disk is precisely analogous to writing it. The command is encoded as 20 in **RKCS1**.

Address Relocation This note will be comprehensible only after reading the next section, but it is included here for completeness.

The contents of register **RKBA** define the low-order 16 bits of the *UNIBUS* address at which data transfers to or from the disk are to begin. Bits 8 and 9 of **RKCS1** are set by the program to the desired bits 16 and 17 of the *UNIBUS* address for data transfers. If bits 8 and 9 of **RKCS1** are clear, **RKBA** still defines a physical address (in which bits 16 and 17 happen to be zero), not a virtual address.

Thus, to prepare to read a word from the disk to the terminal printer buffer (an improbable notion), setting **RKBA** to contain 177566 is not adequate. We must also set bits 8 and 9 of **RKCS1**. In general, these bits will be dealt with by a routine that, by referencing the relevant page address register, performs a software version of a memory management function.

Interrupts The *RK06 interrupt enable bit* is bit 6 of **RKCS1**. It does not work to issue a command and set the interrupt enable bit later, even if the drive is still performing the command when the bit is set. Instead, the program should set interrupt enable when issuing a command. Thus

```
MOV      #123,RKCS1
```

is: write and go, and interrupt when the command is complete.

The **RK06** disk has interrupt priority 5 and traps through location 210. Location 210 must have been set up to point to the user's interrupt service routine; and location 212 must have been set up to give the priority level at which the CPU is to perform the interrupt service routine. For reasons that will appear later, the disk may well interrupt its own interrupt service routine; to avoid losing control, we recommend that location 212 be set to implement at least priority 5.

A considerable complication is that the disk will interrupt under three distinct circumstances: drive ready, controller ready, and when an error has occurred.

Drive and Controller Interrupts Both drive and controller can interrupt when their respective tasks are completed. Once the controller has interrupted, the program may start loading device registers ready for the next command, but the command must not be issued until the drive is ready. Thus it may be important to be sure that the drive rather than the controller has caused an interrupt. A program can do this by checking bit 14 of **RKCS1**, which is set when the drive interrupts. The drive interrupt bit may be cleared, ready for the next interrupt, by issuing the "drive clear" command, function code 4 in **RKCS1**; remember that the go bit must also be set.

It is pointless for the interrupt service routine to go into a loop that tests the drive interrupt bit until it is set. If the controller has interrupted and the interrupt service routine runs at high priority, the drive would have no way of asserting its interrupt; thus the loop would be perpetual. The interrupt service routine could, however, respond to discovering that the interrupt was controller generated by going into a loop to check the drive ready bit in **RKDS**.

Alternatively, the program can maintain its own memory location **DSKRDY** and ensure that, say, it is clear when the disk drive is occupied with a command and set to -1 if the drive is ready

to accept a new command. If this is done, subroutine **CHKDSK** should now begin with

```

CHKDSK:   TST      DSKRDY
          BPL      CHKDSK

```

This loop will terminate when the disk is ready, and **CHKDSK** can proceed with the sequence of checks discussed earlier. If these are satisfactory, **CHKDSK** will return and the program can issue a command to the disk. Before issuing the command, however, the program should **CLR DSKRDY** to indicate that the device is now busy.

When the drive is ready, it will interrupt the loop at **CHKDSK**; the interrupt service routine will then set **DSKRDY** to indicate that the drive is available, and will then return.

```

SERV:     BIT      #40000,RKCS1      ;who interrupted?
          BEQ      1$               ;controller: wait for drive
          DEC      DSKRDY           ;drive interrupted
          BIC      #100100,RKCS1    ;disable interrupts
1$:       RTI

```

(Why does this routine not check whether **DSKRDY** is already set to -1 ?) The interrupt service routine should, as indicated, disable further interrupts when it returns from a drive interrupt. The natural **BIC #100,RKCS1**, however, has an unfortunate side effect. For technical reasons, it actually also *sets* the *controller clear bit*, bit 15 of **RKCS1**. Setting this bit under program control clears all the **RK06** device registers (except bits indicating drive errors), so all record of current data transfer locations would be lost. This is avoided by specifically demanding that the controller clear bit should be cleared, as in the preceding routine.

- EXERCISES:**
- (i) Draw a flowchart for a program to control the disk using a subroutine such as **CHKDSK** and an interrupt service routine.
 - (ii) Write a program to start the spindle when it is stopped; then stop it ("unload" command, function code 6 in **RKCS1**); then start it again.
 - (iii) Write your own macros to start and stop the disk spindle. (If either command is issued when the spindle is already in the required state, the controller will interrupt, but the drive will not.)

Error Conditions We have saved the most troublesome problem of storage device programming until last. Unfortunately, it is not possible to give more than cursory treatment of this topic in the space available.

When an error occurs, the controller will set a bit, specific to that error, in one of the device registers. The controller will also set bit 15 of **RKCS1** which, as well as being the controller bit, is also the *combined error bit*. If interrupts have been enabled, the controller will now interrupt.

Thus, the interrupt service routine should, before doing anything else, check whether the interrupt was caused by an error condition.

```

SERV:     TST      RKCS1
          BMI      IOERR            ;error routine

```

The responsibility for some errors will lie with the programmer. For example, the controller may have reached the end of sector 21 on surface 2 of cylinder 410 during a data transfer, with data still remaining to be transferred. This will set bit 9 in the *drive error register*, **RKER = 177454**.

Many errors, however, can occur as the natural result of an attempt to establish communication between two devices as sophisticated, and as disparate, as disk and memory. In particular, device timing errors are common. If a check of the error bits reveals that such a "hardware error" set the

combined error bit, the best thing to do is try the operation over again. For this, the program should clear the combined error bit; remember that this will clear the device registers so, if their contents will be needed, they should first be saved. The program should then issue the command **RESET** to initialize the UNIBUS, and branch back to repeat the I/O operation.

- EXERCISES:** (i) Use the manual for your system's storage device to produce a complete list of the possible error conditions and their corresponding error bits.
*(ii) Write a program to move data from one memory block to another, via the disk.

4.5 MEMORY MANAGEMENT

The familiar procedure of interpreting the contents of a PDP-11 word as an address enables us to reference byte locations numbered 0 to 177777: to a total of $2^{16} = 65536$ bytes. It is customary to use the notation K for $2^{10} = 1024$ (= 0 2000) when describing quantities of memory. Thus a PDP-11 word gives us the capability of addressing 64K bytes, or 32K words, of memory.

Every PDP-11 processor possesses, as standard equipment, a certain amount of memory, with system owners able to purchase more than the basic allotment if desired. It would be reasonable to suppose that the maximum amount of memory that could be installed on a PDP-11 computer would be 32K words; there would simply be no way of addressing any more. Indeed, since 4K word addresses are reserved on the UNIBUS for I/O device registers and other locations used for control functions, it would seem that a PDP-11 could not support more than 28K words of memory. On the smallest members of the PDP-11 family, this is indeed the case.

In the larger PDP-11s, however, the internal addressing registers of the CPU contain eighteen bits, and UNIBUS addresses are also eighteen-bit. Thus the CPU can reference, and the UNIBUS can support, up to $2^{18} = 256K$ byte locations, or 128K word locations. With 4K words reserved for I/O device registers and the like, this leaves a potential of 124K words of memory. While this might seem to be a vast amount of memory space, several users (especially extravagant ones) in a timesharing environment can readily exhaust it.

This memory, however, is still composed of familiar sixteen-bit PDP-11 words, so there would still seem to be an intractable problem, in that the contents of a single sixteen-bit word cannot supply an eighteen-bit address. Indeed, if the user takes no action to ensure otherwise, only 32K word locations on the UNIBUS will ever be accessed. Of these, 28K will be memory locations. Since $56K = 0\ 160000$ (we are counting bytes), a program can refer to memory word addresses 0 to 157776 in the usual manner. Such a reference is interpreted by the CPU in the obvious way, as designating the UNIBUS location bearing the specified number. Thus **CLR @#100000** clears the UNIBUS location numbered 100000; this location is a memory word. We prefer in this section to use absolute addressing, since the address then appears explicitly in the coding.

The I/O device registers, however, always have UNIBUS addresses in the 4K highest-numbered locations; these are the eighteen-bit UNIBUS addresses 760000 to 777776. For example, the terminal printer buffer occupies UNIBUS location 777566. This number is too large to squeeze into a PDP-11 word, so an instruction such as

```
MOVB    #102,@#777566
```

cannot be encoded without modification. We have, however, already seen that

```
MOVB    #102,@#177566
```

references the terminal printer buffer. It does so because the CPU automatically *relocates* all program address references in the range 160000 to 177776, interpreting them as the low-order sixteen bits of a UNIBUS address in which bits 17 and 18 are both set to 1.

It is crucial not to confuse this form of address relocation with the program relocation performed by the linking loader. The task of the latter is to ensure that, regardless of where a program is loaded into memory, the result of the effective address calculations performed by the CPU lead to the locations we wanted. For example, if we write **MOVB #102,177566**, some work is needed on the part of the linking loader to ensure that the location referenced is the same as if we had written **MOVB #102,@#177566**. In each case, program address 177566 is referenced. The function of the form of relocation introduced in this section is now to associate this address (the *virtual* address) with the UNIBUS location to be referenced (the *physical* address). Note also that the conversion of virtual to physical addresses is performed by hardware.

We see that for program addresses 0 to 157776, no relocation has been performed; the physical address is the same as the virtual address. For virtual addresses 160000 to 177776, however, the physical address is obtained by putting the virtual address in the low-order sixteen bits and setting bits 17 and 18 to 1.

PDP-11s supporting more than 28K words of memory have special provisions for controlling the amount of relocation. With proper choice of virtual address and amount of relocation, any physical address may then be referenced. The hardware by means of which relocation is controlled is called a *memory management unit*. On some PDP-11s it is an integral part of the CPU, on others it is available as an optional accessory, while on the smallest models it is unavailable.

The various CPU models perform memory management functions at different levels of sophistication and complexity. Our discussion will center on the 11/34, where memory management is integral to the CPU, indicating occasionally how other models differ.

Memory Pages On the PDP-11/34, virtual address space is divided into eight *pages*, each of which will be relocated as a whole by memory management. There are 32K words of virtual, or program, addresses, with each page being 4K words in length.

The assignment of virtual addresses to pages is predetermined and can in no way be changed. Specifically, we have:

<i>Page number</i>	<i>Virtual address range</i>
0	000000-017776
1	020000-037776
2	040000-057776
3	060000-077776
4	100000-117776
5	120000-137776
6	140000-157776
7	160000-177776

For each page there are two special UNIBUS locations, the *page address register* and the *page descriptor register*. We use for these the mnemonics KPAR0 to KPAR7 and KPDR0 to KPDR7 (the reason for the K will be explained later). The UNIBUS (physical) locations for these registers are:

<i>Page number</i>	<i>KPAR</i>	<i>KPDR</i>
0	772340	772300
1	772342	772302
2	772344	772304
3	772346	772306
4	772350	772310
5	772352	772312
6	772354	772314
7	772356	772316

If the 11/34 is being operated as a standalone machine, these locations may be read, and certain bits of them written, from the console. They may also be accessed by a program, as long as the program has some route to these physical addresses. We saw earlier that even if nothing is done about memory management, page 7 addresses are always relocated by the hardware in such a way as to make these physical addresses accessible.

If, however, the 11/34 is being operated as a timesharing system and you are not a privileged user, you are unlikely to be able to access these locations. Later in this section we shall study the mechanism that prevents your doing so.

Enabling Memory Management When the computer is first switched on or when the CPU is first restarted after a HALT, memory management is inactive. The unit is switched on by setting the *memory management enable bit*; this is bit 0 in the *memory management status register number 0*, mnemonic SR0, UNIBUS location 777572.

Observe that with memory management disabled, the automatic relocation of page 7 addresses lets us reference SR0 as virtual address 177572. Thus

```
SR0 = 177572
...
BIS      #1,@#SR0
```

enables memory management.

Consider now the following sequence:

```
SR0 = 177572
KPAR7 = 172356
...
CLR      KPAR7
BIS      #1,@#SR0
BIC      #1,@#SR0
```

Clearing KPAR7 will not be necessary if the computer has just been switched on, since this location will be clear anyway. But if the CPU has been halted after an operating system has been in residence, although memory management will be disabled, the memory management registers will still be as set by the operating system. Memory management will normally set up KPAR7 so that the usual relocation of page 7 addresses to UNIBUS I/O device register addresses still takes place. This requires a particular setting of KPAR7, which we destroy by clearing that location.

Remember, however, that we are clearing KPAR7 while memory management is still inactive. Thus, in the next line, our reference to SR0, virtual address 177572, is automatically relocated by the hardware to UNIBUS location 777572. Thus, the effect of the BIS instruction is, indeed, to enable memory management.

Next we try to disable memory management again, with the BIC instruction. This would certainly be the effect of clearing bit 0 of UNIBUS location 777572. Our reference in the BIC instruction, however, is to virtual address 177572. By enabling memory management in the previous instruction, we have, of course, disabled the automatic relocation of virtual address 177572 to physical address 777572. Instead, the hardware, noting that 177572 is a page 7 virtual address, goes to KPAR7 to determine how it should relocate it. We have deliberately refrained from setting KPAR7 to accomplish the proper relocation. Hence our BIC instruction will reference some other physical location than the memory management status register, so does not disable memory management.

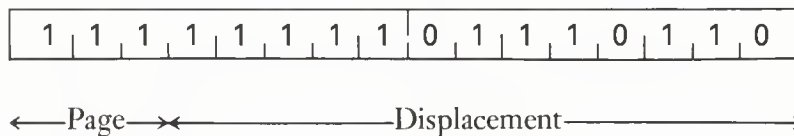
Nor is there any way in which the program can remedy the situation. It cannot now set up KPAR7 to achieve the correct relocation, because it has no way of referencing KPAR7! All the I/O device registers are similarly outside the addressing range of the program. The only way to disable memory management after this is to halt the CPU.

Creation of a Physical Address With memory management enabled, every program address reference is broken down by the hardware into two sections. Bits 13 to 15 are used to determine the page number of the virtual address; this is the *active page field*. Suppose the virtual address is 177566; this has 111, the binary code for 7, in bits 13 to 15, so page 7 is indicated. The hardware therefore goes to KPAR7 to discover the *base address* for page 7. It determines this from bits 0 to 11 of the page address register (the other bits are not used on the 11/34). This 12-bit field in KPAR7 is shifted 6 bits to the left to create the physical base address for page 7.

The remaining 13 bits, bits 0 to 12, of the virtual address constitute the *displacement field* within the page. This number is added to the base address to form the physical address to which the virtual address will be relocated.

Note that the entire relocation procedure is performed by the hardware. All the programmer has to do is set up the memory management registers.

Normally, an operating system will set KPAR7 to 7600. Suppose again that the virtual (program) address is 177566. This divides into the two fields

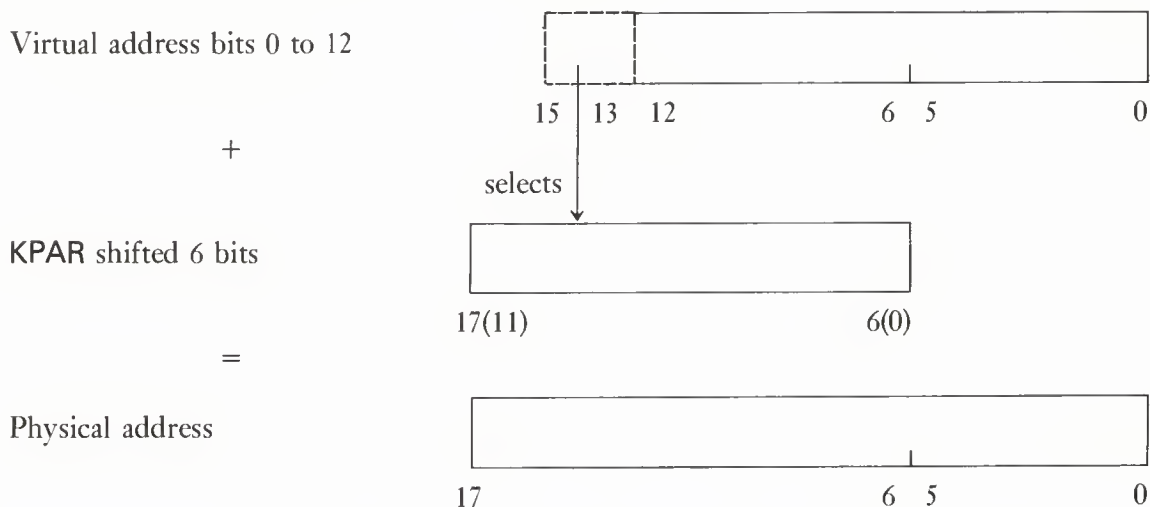


with page field 7 and displacement field 17566. Note that the displacement field always gives a (byte) address between 0 and 17777; a page is 8K (= 0 20000) bytes, 4K words, in length. The contents of KPAR7, 7600, are now shifted 6 bits, to give 760000 as the base address for the page. So the physical address to which virtual address 177566 will be relocated is

Base address	760000
+ Displacement field	+ 17566
Physical address	777566

as is required to reference the terminal printer buffer.

The process may be viewed diagrammatically as follows.



Note that bits 0 to 5, the two lowest-order octal digits, are always the same in the physical as in the virtual address. For a given virtual address, incrementing its KPAR by 1 adds 100 to the physical

address. The quantity 100 bytes is thus the smallest unit of relocation and is called a *block* for memory management purposes.

Page Length and Access A virtual memory page always consists of 200 blocks. The memory management unit will relocate the first block of a page, computing the base address as just described. It will, however, refuse to relocate any more blocks unless specifically instructed to do so. Thus program references to locations 000100 to 017776 would fail, because there would be no translation of these virtual addresses into physical addresses. References to locations 020000 to 020076, however, would work.

To determine how many blocks of a given page of virtual addresses should be relocated into physical addresses, memory management refers to the appropriate page descriptor register. The high byte of this is set by the programmer to the number of blocks (after the first) for which relocation is desired. Thus

```
BIS          #77400,@#KPDR7
```

will inform the memory management unit that all 200 blocks of page 7 addresses are to be relocated into physical addresses; it specifies that the *length* of page 7 is to be 200 blocks.

Bits 1 to 2 of the KPDR for each page must be set by the programmer to determine the CPU's *access rights* to the page. If both of these bits are set to 1, all program instructions referencing virtual addresses in the page are allowed; this is a *resident, read/write* page. If bit 1 is set but bit 2 is clear, program instructions are allowed to reference virtual addresses in the page, as long as they do not attempt to change their contents; this is a *resident, read only* page. If both of these bits are clear, a program may make no references to the page; it is then *nonresident*.

Thus the sequence

```
MOV          #7600,@#KPAR7
BIC          #6,@#KPDR7
BIS          #1,@#SR0
```

effectively cuts off communication with the I/O devices (unless another page can already reference them). KPAR7 is correctly set up for relocation of page 7 addresses to I/O device register addresses, but KPDR7 is set to make the page nonresident. Since all the memory management registers are in the same UNIBUS region, it is not now even possible to point another virtual address page to the same region. Once again, the only escape is to halt and restart the CPU.

Note that replacing #6 by #77400 in the BIC instruction has an equally drastic effect. Only one block of page 7 will be relocated, so UNIBUS addresses above 760076 are totally inaccessible.

We can, of course, relocate any virtual address page we please to the I/O device register region on the UNIBUS. To do this, we must set up the page registers; since these are in the UNIBUS region in question, we must start either with memory management disabled, or with the device registers accessible through some other page.

Let us write another program to print out the letter B, this time by addressing the terminal printer buffer as location 34566 within the program. Virtual address 34566 breaks down into page address field 1 and displacement field 14566. Thus, to achieve a relocation of virtual address 34566 to physical address 777566, we must set KPAR1 to provide a base address as follows.

Physical address	777566
– Displacement field	– 14566
Base address	763000

That is, KPAR7 must be set to 7630. The entire program, starting with memory management disabled, is

```

                KPAR1 = 172342
                KPDR1 = 172302
                SR0 = 177572
                TPB = 34566                                ;!
START:          MOV      #7630,@#KPAR1
                MOV      #77406,@#KPDR1
                BIS      #1,@#SR0
                MOVB     #102,@#TPB
                HALT
                .END      START

```

- EXERCISES:**
- (i) What is the smallest number that this program could specify in the high byte of KPDR7?
 - (ii) Write a routine to make virtual addresses 17000 and 20000 relocate to the same physical address; restrict the use of 17000 to read only references.

Protection Violations The limits on page lengths and access rights that can be set in the page descriptor registers are used in timesharing systems to protect different users' memory allocations from each other and to protect the operating system from all nonprivileged users.

An instruction that would violate any of these restrictions will not be carried out; the memory management unit will *abort* the instruction. The unit will also cause a program interrupt (regardless of CPU priority level), using a two-word vector at location 250. In this connection, note that *all interrupt and trap vector addresses are virtual address locations*. Thus

```
MOV      #SERV,@#250
```

will always set location **SERV** as the address of the program's memory management interrupt service routine. When an interrupt occurs, PC will not be loaded with the contents of UNIBUS location 250; rather, since memory management is operative, *virtual* address 250 will be relocated and PC will be loaded with the contents of the resulting physical address.

- EXERCISES:**
- (i) What happens if a program changes the contents of KPAR0 after setting up interrupt vectors?
 - (ii) Write a routine at the end of which it is possible to instruct

```
MOV      #SERV,@#55550
```

to set up the memory management interrupt vector. What happens if the program subsequently changes KPAR2 (but not KPAR0)?

The address given in a program as the start of an interrupt service routine is itself a virtual address, so it is subject to relocation. Suppose that a program makes page 3 nonresident, enables memory management, and continues with

```
MOV      #60000,@#250
TST      @#60000
```

The first instruction is acceptable. Although the number 60000 occurs in it, no reference is made to the contents of virtual address 60000, so there is no memory protection violation. Thus, this instruction successfully sets up location 60000 as the address of the memory management interrupt service routine.

The second instruction tries to reference the nonresident page 3. The memory management unit therefore aborts the instruction and traps to virtual location 250. The CPU then attempts to load PC with the contents of virtual address 60000 in order to transfer control to the interrupt service routine. This, however, again violates memory protection; another interrupt occurs, and so on. The result is a perpetual loop.

EXERCISE: A program sets location 250 to contain 60000, makes page 0 nonresident, enables memory management, and continues with TST @#250. What happens?

Processor Modes The difference between a privileged and a nonprivileged user in a time-sharing system is to be found in the *mode* in which they may operate the CPU. On the 11/34 there are two modes: *kernel* and *user*. When the computer is first switched on, it runs in kernel mode and will continue to do so while an operating system program, or that of a privileged user, is running. For a nonprivileged user, however, the operating system will change the CPU mode to user after taking steps to ensure that the user has no way of changing the mode back.

The most obvious difference between kernel and user modes is that when the CPU is running in user mode, the HALT instruction is illegal. Instead of stopping the CPU, it will cause a trap to location 4 or 10. (Which location depends on the CPU; the manuals are unreliable here, so best check directly. We have even encountered an RSX-11 operating system that printed out a "system crash at location 4" message when, on an 11/34, the trap had actually been to location 10.)

The mode is determined by the state of bits 14 and 15 of the processor status word, PS. Note that this is UNIBUS location 777776. When both of these bits are clear, the CPU is running in kernel mode. With both bits set, user mode is operative. It is illegal on the 11/34 to have these two bits set differently, but it is meaningful on larger PDP-11s with more than two modes of operation.

On the 11/45 and larger models, a program running in user mode is prevented by the hardware from changing the mode bits in PS. On the 11/34, however, the only way of preventing user-mode programs changing to kernel mode is by using memory management protection to render PS inaccessible to user-mode programs. This is possible because there are, in fact, two complete sets of page address and descriptor registers: one set for kernel mode and one set for user mode.

Our description so far, including the UNIBUS locations for the registers, has been of kernel space memory management. The operation of user space memory management is essentially similar. The UNIBUS addresses are:

<i>Page number</i>	<i>UPAR</i>	<i>UPDR</i>
0	777640	777600
1	777642	777602
2	777644	777604
3	777646	777606
4	777650	777610
5	777652	777612
6	777654	777614
7	777656	777616

When the memory management unit is presented with a virtual address, it first checks PS to determine the mode of operation. This tells it which set of registers it should use when going through the relocation process described earlier. The contents of each set of memory management registers are

wholly independent of the other. Thus it is feasible (and, in fact, normal) for user-mode programs to be denied access to the UNIBUS region containing the I/O registers.

Nevertheless, a program running in user mode must be able to perform I/O. We already know that this is done with traps to operating system routines and are now able to examine the process more closely. Let us, for the last time, write a program to print out the letter **B** at the terminal. We shall let a user-mode program achieve this with the **TRAP** instruction. Starting in kernel mode, we must set up the trap vector.

```
MOV      #TRP,@#34
MOV      #340,@#36
```

We have chose location **TRP** for the start of the trap handler and have set priority 7 so that we can forget about device interrupts. Note that all interrupt and trap vector locations are numerically fixed in *kernel* virtual address space. They will be accessible to user-mode programs only if some page in user virtual address space accesses the same physical locations as do these low addresses in kernel virtual address space. Normally, however, memory management is used to deny user-mode programs access to these vectors.

At location **TRP**, we have

```
TRP:      MOVB      #102,@#TPB
          RTI
```

Of course, the symbol **TPB** will be defined elsewhere; assuming the usual setting of **KPAR7**, we shall have **TPB = 177566**.

The loader will assign some virtual address to **TRP**. Memory management may well have been used to ensure that the physical location to which **TRP** is relocated is inaccessible to user-mode programs. At the very least, it will have been made read only. In the latter case, a user-mode program could perform **JMP TRP** without violating memory protection. Presumably, however, the device I/O registers will have been denied to user programs, so performance of the instruction at **TRP** would be aborted.

EXERCISE: Write the complete routine so far, including setting up the memory management registers to protect vectors, device registers, and kernel programs from user programs.

Our program can now set user mode with

```
BIS      #140000,@#PS
```

assuming that **PS** has been defined so as to relocate to UNIBUS location **777776**. After this, all attempts to perform I/O directly will trap to location **250** in kernel virtual address space. We are not writing an interrupt service routine, but should at least control the interrupt by setting the contents of location **250** equal to **252** and clearing location **252**. (What happens then?)

The program can, however, issue the instruction **TRAP**. This puts the contents of **PS** and **PC** onto the stack and loads **PC** from location **34** and **PS** from location **36** in kernel virtual address space. Now bits 14 and 15 of this location **36** are clear, so *the new contents of PS will specify kernel mode*. Thus the **MOVB** instruction at location **TRP** is performed, and **B** is printed out at the terminal. The **RTI** instruction now reinstates the former contents of **PC** and **PS**, *returning to user mode in the process*.

EXERCISE: Write the complete program.

In practice, it would not be helpful for a timesharing operating system to respond to all user requests for output by referencing the terminal printer buffer at UNIBUS location 777566, because this is at the *console terminal*, which is found alongside the CPU in the computer room. Communication between the CPU and the various user terminals is, in fact, performed by addressing registers in a complex interfacing device.

Communication Between Spaces In Section 3.4 we considered how an operating system EMT routine might pick up an argument from the low byte of the EMT instruction. Suppose now that the EMT instruction was issued by a user-mode program. The EMT trap handler saves the contents of R0 to R5 on the system stack; the return PC is then at 14(SP). This is brought into R0.

```
MOV      14(SP),R0
```

The EMT instruction itself is now at the virtual address given by $-2(R0)$ but in *user* virtual address space. The EMT trap handler, however, will be operating in kernel mode. Thus suppose that the EMT instruction had been loaded into user virtual address 2000. If the trap handler continued with

```
MOV      -(R0),-(SP)
```

it would put onto the stack the contents of virtual address 2000 in *kernel* space. This would be the same location only if user and kernel page 0 had been equally relocated.

Instead, the trap handler should use the Move From Previous Instruction space MFPI instruction. This is a single-operand instruction. It interprets the result of its effective address calculation as a virtual address in the previous space which, in the present case, was user space, and puts the contents of that address onto the system stack. Thus the trap handler should continue with

```
MFPI     -(R0)
```

Communication in the other direction, from the system stack to an address in the previous space, is similarly accomplished by the Move To Previous Instruction space MTPI instruction.

Previous mode is defined as being the mode in which the CPU was operating before the last trap or interrupt. When a trap or interrupt occurs, the second word of the vector is used to load all but bits 12 and 13 of PS; these bits are loaded with the previous contents of bits 14 and 15 of PS. The MFPI and MTPI instructions check bits 12 and 13 of PS to determine which virtual address space to reference.

- EXERCISES:**
- (i) Expand our program above so that TRAP 'X can be used with any letter of the alphabet to print out that letter at the terminal.
 - *(ii) Write your own macros to perform MFPI and MTPI without using these instructions.

System Stacks For each of its operating modes, the CPU has a separate stack pointer. These are quite distinct CPU registers, with each being set wholly independently of the other. However, a program cannot directly choose to reference one stack pointer rather than the other; the choice is prescribed by the mode in which the CPU is running.

Thus, if a program has defined $SP = \%6$, any reference to SP will be interpreted by the hardware as kernel SP when the CPU is in kernel mode and as user SP when the CPU is in user mode. Normally, it is of no concern to the user that the operating system uses a different stack pointer.

When a trap or interrupt occurs, the CPU checks bits 14 and 15 of the second word of the vector to determine on which stack to store the return PC and PS. This ensures that RTI will make the correct

return. Clearly, a trap or interrupt service routine should not change its own mode before returning, because then **RTI** would reference the wrong stack. Note that **MFPI** and **MTPI** transfer information between previous space and the stack for the current operating mode of the instruction.

Some operating system macros have the user program pass argument addresses on the stack. It requires two instruction steps for a kernel-mode routine to pick up an argument from the user stack. First, the address of the argument, equal to the contents of the user stack pointer, is put onto the kernel stack.

MFPI **SP**

Note that when an **MFPI** or **MTPI** instruction references **SP** in *addressing mode 0*, the CPU interprets the reference as previous mode **SP**.

Since a pointer to the desired argument is now on the kernel stack, the argument itself is referenced **@(SP)**, via the *kernel* stack, but in *user* virtual address space. We replace the address of the argument with the argument itself on the kernel stack by

MFPI **@(SP)+**

using the same addressing trick as for coroutine control transfers. Note that when an **MFPI** or **MTPI** instruction references **SP** in *any addressing mode except 0*, the CPU interprets the reference as current mode **SP** but interprets the resulting effective address in previous space.

- EXERCISES:**
- *(i) Write an instruction sequence for a kernel-mode program to put an argument value onto the user stack.
 - *(ii) Amend your **MFPI** and **MTPI** macros to take the two-stack situation into account.

A ODT

ODT is the On line Debugging Technique of DEC PDP-11 operating systems. As with any systems program, the implementation may vary from one operating system to another. The basic facilities of ODT will, however, be much the same, in spite of variations in the form of commands.

RUNNING ODT

The ODT program normally exists as a relocatable binary file **ODT.OBJ**. This must be linked with the program to be debugged. Suppose that we want to use ODT on the program **TEST.MAC**. We must first compile **TEST.MAC** and then use the linking loader to *link* **TEST.OBJ** with ODT.

```
TEST,TEST=ODT,TEST↵
```

This provides the linking loader with two input files. It fuses these together and creates two output files: **TEST.SAV** and **TEST.MAP**. Note that these files give the memory image form of **TEST** linked together with ODT; if you want to keep your memory image file of **TEST** itself intact, you can just specify different names for the output files when you link with ODT.

Depending on the linking loader, it may be necessary that, as above, ODT appears before the program to be debugged in the list of input files. The reason would be to ensure that when the monitor level command

```
RUN TEST↵
```

loads the linked files into memory, it transfers control to the start address of ODT rather than to that of **TEST**. When this happens, ODT types a brief message, giving the version number of the program, followed by a * to indicate that it is ready to receive commands.

You should determine how to reach this point in your own system before proceeding. The normal way of returning to monitor command level from ODT is to type **^C** when ODT is expecting a command.

GLOBAL SYMBOLS

One use of ODT is to follow through changes to the contents of memory locations crucial to the program at various stages of execution. Commands to ODT must, however, refer to numerical memory addresses and not to names associated with them by the programmer. Thus we cannot look at the contents of **MEM** until we know where **MEM** has been loaded. The **.MAP** file will not tell us this, nor will it even tell us the load address for the start address of our program, since our program is now a mere part of a structure whose start address is that of ODT.

We can, however, induce the linking loader to recognize the existence of our program's start address when it creates the .MAP file. For any symbol in the program, this is done by declaring in the program with the .GLOBL directive that the symbol is to be *global*, or available for reference from outside the program.

```

                .GLOBL    START
START:          ...
                ...

```

This directive is also necessary when different subroutines of a program are compiled separately and joined by the linking loader. Suppose we are going to link MAIN.MAC and SUB.MAC after compiling them and that SUB.MAC incorporates subroutine SUBRTN. If MAIN.MAC includes a call on SUBRTN, the statement .GLOBL SUBRTN must appear in *both* MAIN.MAC and SUB.MAC.

Note that separately compiled parts of a program, or *modules*, must each have an .END statement. However, only the module containing the start address for the whole should include it in the .END statement. All the others should terminate with just

```

                .END

```

It is not generally necessary to remove the start address from the .END statement of a program to be linked with ODT, since most operating systems make provisions for ensuring that control passes to ODT.

PROGRAM EXAMINATION AND CORRECTION

Before linking a program with ODT, you should therefore declare its start address to be a global symbol. In addition, *you should arm yourself with a printout of the listing file for your program* (without ODT).

After linking your program with ODT, exit from the linker program and look at the .MAP file. Make a note of the load address for your program's start address. Now run the linked program, so that ODT takes control and prints a *.

Suppose, for example, that your program listing shows MEM as location 162. This, of course, is relative to your program's start address. So if the .MAP file showed, say, 6316 as the load address for your program's start address, you could reference MEM when using ODT as location 6500 (= 6316 + 162).

Since it is inconvenient to do this relocation oneself for every location reference, the first command to ODT should tell it to relocate all our subsequent address references. ODT maintains a table of eight *relocation registers*, numbered 0 to 7. This allows independent relocation of up to eight separate program modules. We just want to supply a single relocation constant of 6316 for the whole program so, in response to ODT's *, we type

```

6316;R

```

(do not follow this with ←), to set the contents of relocation register 0 to 6316. ODT will respond with a fresh *. After this, whenever ODT types an address, it will precede it with 0, to indicate that it is relative to the contents of relocation register 0. You also must refer to the relocation register you are using to pass an address to ODT, as shown later.

We now have the great convenience of being able to specify to ODT exactly the addresses that

appear in our listing file. Suppose that our program had **CLR R1** at location **72** relative to the start address. If we type

0,72/

(do not follow this with $\leftarrow$), ODT will respond, on the same line, with,

005001

The symbol **/** refers ODT to the word whose address has just been typed and instructs it to type out the contents of the word. It is important to realize that ODT has done two things before typing out the contents of the location: it has set its location indicator, or *pointer*, to reference relative location **72**, and it has *opened* that location.

Once a location has been opened, its contents may be changed. Suppose you realize that the instruction should have been **CLR @R1**. Then, after ODT has typed out **005001**, you type in the entire new contents; that is, you type

005011

and this time you do follow it with a $\leftarrow$. The $\leftarrow$ instructs ODT to *close* the word, with the new contents. If you are happy with the contents of a location as typed out by ODT, just type in a $\leftarrow$. If no new contents have been specified, ODT will close the word, leaving the contents unchanged.

The changes you make in your program while using ODT are not preserved in your file; they are lost as soon as you exit from ODT. Consequently, you should make notes on a copy of your program listing while using ODT.

If you make a typing error when entering a command to ODT, you cannot just RUBOUT the offending character. Pressing the RUBOUT key is the correct step to take, but it will cause ODT to ignore the entire command and type a fresh *. You should then enter the command again.

Although ODT closes a word in response to $\leftarrow$, it does not move its pointer. Typing **/** will reopen the location currently referenced by the pointer and type out its contents. This is convenient for verifying any changes you have just made.

After, if you so wish, modifying a word, you may close it not only with $\leftarrow$, but also with $\downarrow$ (LINE FEED), $\wedge$, $\leftarrow$, $@$, $>$, or $<$. Note that the $\leftarrow$ character is $\_$ on some terminals.

When working through a program line by line, $\downarrow$ is the most useful word-closing instruction: it also moves the pointer on to reference the next word location, opens it, and types out its contents. Suppose, for example, that the instruction following the **CLR R1** at location **72** is **MOV R3, MEM**. Closing location **72** with $\downarrow$ instead of $\leftarrow$ will then cause ODT to type out

0,000074 /010367

Note the indication that the address is **74** relative to the contents of relocation register 0. Location **74** is now open; we may change its contents or close it in any way, as we please.

The word-closing instruction $\wedge$ goes in the direction opposite to $\downarrow$. It sets the pointer to reference the previous word, opens that word, and types out its contents.

If **MEM** was at location **162** in the program, closing location **74** with $\downarrow$ will produce

0,000076 /000062

Your listing file will confirm that this is the correct reference to **MEM** in a relative mode instruction. The fetch from memory will be performed with **PC** pointing one word ahead of relative location **76**; so the location referenced will be relative $(76 + 2) + 62 = 162$.

Closing a word with $\leftarrow$ causes ODT to index the contents of the word being closed by the address of the next word; as we have just seen, this gives the address of the word referenced in a relative mode instruction. ODT then opens that word and prints out its contents. Thus, in this case, closing location 76 with $\leftarrow$ would result in ODT printing out

0,000162 /000000

assuming that the program had been assembled with **MEM** containing zero. This enables us to follow through the effect that such an instruction will have when it is executed.

Note that there is no single ODT instruction to show us where indexing with any register other than **PC** will point. The $\leftarrow$ instruction itself does not reference **PC**; indeed, the contents of **PC** would point to the routine in the ODT program carrying out the $\leftarrow$ instruction. Rather, $\leftarrow$ uses ODT's pointer to determine where relative addressing will lead at execution time.

If the instruction being investigated had instead been **MOV R3,@#MEM**, opening location 76 would produce

0,000076 /006500

with direct reference to the load address of **MEM**. To look at the contents of **MEM**, this time we should close location 76 with the **@** command. ODT would respond with

0,000162 /000000

treating the contents of the location being closed as the address of the next location to be opened. Of course, since we have neither run the program nor put new contents into **MEM** with ODT, it will contain just what our program specified in the **.WORD** directive for **MEM**.

Note that if we modify the contents of a location before closing it with $\leftarrow$ or **@**, the new contents will be used to determine the address of the next location to be opened.

An $\leftarrow$ or **@** instruction can be used to check through a routine to which the program transfers control with **JSR** or **JMP**, depending on the addressing mode for the destination of the jump. Since the new location opened is the destination of the jump, $\downarrow$ can then be used to examine successive locations in the routine. To examine the destination of a branch instruction, however, we must use the **>** word-closing instruction. Suppose our program had the instruction **BNE MEM** at location 200, as given by the listing file. With **MEM** as location 162, ODT would give us

0,000200 /001370

since the offset is 10 words back from the location following the branch instruction. If now we close location 200 with **>**, ODT will type out

0,000162 /000000

ODT responds to **>** by using the low byte of the word being closed to calculate the offset. It does not trouble to check that the high byte actually encodes a branch instruction. If a word is modified before being closed with **>**, the new contents of the low byte will be used to determine the next location to be opened.

You may realize while debugging a program that a branch or relative mode instruction references the wrong location. Suppose that you want to change the instruction at location 200 to **BNE LABEL**, where **LABEL** is location 316. First, open location 200 with the **/** command

0,200/

so that ODT types out

```
001370
```

now use the ;O command to have ODT calculate the offset to location 316, by typing in

```
0,316;O
```

ODT will respond by typing out

```
000114 046
```

giving both the byte count offset for use in a relative mode instruction and the word count offset for use in a branch instruction. ODT will not print the word count offset if the location is beyond the range of a branch instruction.

We want to amend a branch instruction, so we just type in

```
001046<↓
```

to effect the desired change and close the word. Note that it would not do to type in just 46<↓ (although 1046<↓ would be in order).

The ←, @, and > word-closing instructions depart from a sequential examination of program locations. ODT will, however, keep a record of the last address being examined before one of these instructions was used to change sequence. The word-closing instruction < will set ODT's pointer to its value before the last ←, @, or > was issued and then act like a ↓. Suppose the instruction JSR PC,PRINT occurs in the program, starting at relative location 220. The relative addressing reference to PRINT is housed at location 222 so, to look through the PRINT subroutine, we close location 222 with ←. Then we use ↓ for sequential checking of locations in the subroutine. When we have seen enough, we close a location with <. This opens location 224, the location that was next in sequence before our diversion to PRINT, and prints out its contents. The < command will not, however, trace our way back through two levels of diversion; the second < will act just like a ↓.

The registers may be examined by preceding the register number with \$ in a / command. Thus

```
$3/
```

will have ODT type out the contents of R3. Register contents may be modified by typing in a new value before closing the register. If a ↓ or ^ command is used to close a register, the next or previous register will be opened. The ← or @ commands may also be used, but > and < should not be used. This restriction is reasonable, since a program cannot transfer control to a register; thus a branch instruction will not be found in a register, nor will a ←, @, or > have accessed it.

ODT's internal register \$S contains the condition codes (in their usual order) as its low-order four bits. These can be examined by typing

```
$S/
```

The other bits of \$S should be ignored.

ODT will also display memory contents as ASCII or radix-50 characters. The \ command is used to open a byte, print out its contents as a number, and then, if possible, as an ASCII character. Thus, if our program set up MEM with

```
MEM:      .WORD      "AB
```

then

0,162/

will have ODT type out

041101

but

0,162\

will result in

101 = A

ODT is now operating in byte mode, and the ↓ and ^ commands will reference successive bytes. Thus ↓ would now produce from ODT

0,000163 \102 = B

The / command with an even address will return ODT to word mode. Note that / with an odd address has the same effect as\.

To examine radix-50 text, use / to have ODT type out the contents of a word as an octal number in the usual way. Then type X (without a ↵), and ODT will display the word contents as three radix-50 characters. However, close the word with ↵ or ↓, because some versions of ODT will endeavor (unsuccessfully) to interpret the other word-closing instructions as new radix-50 word contents for the referenced location.

PROGRAM EXECUTION

The ;G command tells ODT to transfer control to an address in our program. Thus

0,0;G

starts program execution at the first location of our program, assuming that we have properly set ODT's relocation register, as discussed. It is, however, not generally very helpful to issue this command without certain preparatory steps, because the program would just run straight through as if it had been executed without ODT.

A simple way to use ;G effectively is to check whether a program works from a certain point on. Suppose, for example, that you want to begin by checking that the printout routine works properly. After putting test data into any appropriate location, if the printout routine begins at relative location 330, we enter

0,330;G

in response to a prompting * from ODT.

This is still not an efficient way to use ODT, because there will be an exit after every trial run. Before starting the program at any location, you should make sure that execution will stop at a convenient point. This is done by setting *breakpoints*; we shall first discuss how to set and remove breakpoints and then discuss how to use them.

A breakpoint is set at a location by typing the address of the location, then the two characters ;B. ODT will respond with a fresh *. Up to eight breakpoints, numbered 0 to 7, may be assigned in this way. When a breakpoint is set, the smallest available number is assigned to it unless we specifically demand otherwise. For example,

0,620;6B

will set breakpoint number 6 at program location 620, as long as that breakpoint has not already been assigned elsewhere. If breakpoint 6 is already in use, it may be removed by typing, in response to a * from ODT,

;6B

and may now be assigned elsewhere. Note that the command to remove a breakpoint does not specify the current address of that breakpoint. Typing ;B removes all breakpoints.

If you forget where you have put a particular breakpoint, type

\$B/

to see the address of breakpoint 0. Then use ↓ to examine the addresses of the other breakpoints in sequence.

The time for setting or removing breakpoints is before starting execution with a ;G instruction or after ODT has completely performed one of the ;P instructions to be discussed next and issued a * accordingly.

One restriction must be observed. ODT replaces the contents of locations at which a breakpoint is set with the BreakPoint Trap instruction BPT, restoring the original contents when the breakpoint is removed. So a breakpoint should not be set at a location on whose contents another program instruction will rely. Thus, for example, if the instruction JSR PC,@TABLE is going to be performed, a breakpoint may not be set at location TABLE. But setting a breakpoint at TABLE will do no harm if the instruction to be performed is JSR PC,TABLE.

We can begin by setting a single breakpoint at the start address by typing 0,0;B as just described. Now we start execution by typing 0,0;G. Execution of program instructions continues until a breakpoint is reached; control then transfers back to ODT before execution of the instruction at the breakpoint.

In this case, since there is a breakpoint at the start address, ODT will stop before any instructions at all are executed and will tell us what is going on by typing out

B0;0,000000

giving the address at which execution stopped, followed by a *. Now we can work through the program line by line. First, we set *single-instruction mode* with the command

;1S

Actually, any nonzero number can replace 1, but some number is needed to distinguish this command from the distinct command ;S discussed later.

Once single-instruction mode is set, the command ;P instructs ODT to execute the next instruction. If our program started with a two-word instruction, ODT would respond to ;P by executing it and typing

B8;0,000004

giving the address of the next instruction to be executed. If the instruction just executed was a branch whose condition was satisfied, or a jump, ODT will print out the address of the destination; this is correct as the address of the next instruction. Note the reference in single-instruction mode to the mythical breakpoint number 8.

At this stage, we can examine and modify the contents of any locations. Then another ;P will get the next instruction executed, and so on. At any time, instead of typing ;P, we can type ;S to leave single-instruction mode, followed by 0,0;G to start the program again from the start address. The breakpoint set at that address will still be there, and modifications made to the contents of locations will remain in force.

If ;P is issued to execute an EMT or TRAP instruction, not only that instruction, but also the entire trap handler and one program instruction following its return are executed without any break. For reasons that will be clear if you have by now read Chapter 4, there will be difficulties in single stepping in this way a call to a monitor input routine. It is better to use a breakpoint at the instruction following the call, as explained next.

If you enter *n*;P, where *n* is an octal number, ODT will let *n* instructions be executed before it recovers control.

Another way of passing rapidly over a sequence of instructions is to use the ;P command *without setting single-instruction mode* (or after issuing a ;S command to leave it). This causes instructions to be executed, starting from wherever execution last stopped, until an instruction at which a breakpoint has been set is reached. Again, control will transfer to ODT before the instruction at the breakpoint is executed. Nothing will be typed out until the breakpoint is reached (unless in the interim the program performs output, or a program error causes an operating system generated exit and error message).

If intervening instructions require terminal input, the user must type in whatever type and quantity of data the program requires. Be careful not to type characters for input when ODT is expecting an instruction, and vice versa.

For each of its eight breakpoints, ODT maintains a location containing the *proceed count*. When a breakpoint is first set with the *address*;B command, the proceed count for that breakpoint will be 0.

When, during program execution with ODT, a breakpoint is reached, ODT checks the proceed count for that breakpoint. It decrements the proceed count by 1 and checks whether the result is zero or negative. If it is zero or negative, ODT sets it to 1, types its message that the breakpoint has been reached, and waits for the next command.

If, however, the result is strictly positive, ODT returns control to the user program. Thus a user can ensure that a breakpoint is ignored until the *k*th time it is reached by setting the proceed count to *k*. If ODT has stopped at a breakpoint, the command

k;P

where *k* is an octal number, will set the proceed count at that same breakpoint to *k* and also issue a ;P command. Note that 1;P and ;P are identical in effect.

It is possible to set the proceed count at a breakpoint even when the program is not stopped there. At ODT's internal location \$B, as described, is a block of eight locations giving the addresses of the breakpoints. Following this block is a location that, when single-instruction mode is set, contains the address of the next program instruction to be executed. Then comes a block of eight locations containing the breakpoint proceed counts. These may be set directly, using ODT; obviously some careful counting of ↓ commands after \$B/ is required.

B Arithmetic

In previous sections we considered how to perform the basic arithmetical operations on integers, as long as the result would always fit into a PDP-11 word. We now introduce the facilities of MACRO-11 for handling large numbers, and not necessarily integral quantities. We can do no more here than scratch the surface of a vast subject, of which full understanding depends on a substantial level of mathematical competence. Our purpose goes little further than making the arithmetical instructions of MACRO-11 available to the student who has learned or will learn the principles of computational arithmetic elsewhere. We also presuppose greater familiarity with binary and octal arithmetic than is needed in the rest of this book.

MUL AND DIV

The earlier account of these instructions gave a simplified version of their operation. If the instruction **MUL MEM,R** is issued, with **R** an odd-numbered register, the earlier account is accurate. If, however, **R** is an even-numbered register, the result of the multiplication is stored as a thirty-two-bit number in **R** and the next register. Consider, for example,

```
MOV    #1100,R2
MUL    #1000,R2
```

The result, **1100000**, when regarded as a thirty-two-bit number, has a **1** in bits 15 and 18 and zeros elsewhere. The low-order sixteen bits are stored in **R3** and the high-order sixteen bits in **R2**. Thus, in this case, the **MUL** instruction leaves **R3** containing **100000** and **R2** containing **4**.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
31														16	15																0
R2																R3															

The **MUL** instruction treats its operands as signed binary integers, and forms a thirty-two-bit product with bit 15 of the high-order word as sign bit. Note that there is always enough room in a double-length word for the product of two single-word-length integers, so the result is always arithmetically correct.

The **DIV** instruction performs division on a double-length integer stored in an even-numbered register and its successor. Consider

```
MOV    #1,R0
MOV    #100007,R1
DIV    #400,R0
```

The first two instructions store the number **300007** in the pair of registers **R0,R1**. The **DIV** instruction puts the quotient, **600**, into **R0**, and the remainder, **7**, into **R1**.

If the first instruction in this sequence had been `MOV #1000,R0`, division would give a quotient too large for a single word. We would be dividing 200100007 by 400, giving a quotient of 400200. If such an attempt is made, the instruction is aborted and the V bit is set.

EXERCISE: Investigate the behavior of `DIV`, particularly with regard to the remainder, when one or both operands is negative.

MULTIPLE PRECISION INTEGER ARITHMETIC

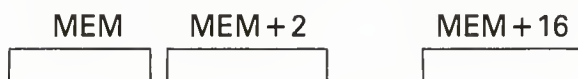
Even quite moderate calculations can produce numbers too large for the confines of a sixteen-bit PDP-11 word. It is therefore quite often necessary to allow a single number to occupy more than one word of memory storage. Two words per number (*double precision*) frequently prove sufficient, but it is barely more difficult to consider the general concept.

Suppose that we want to be able to handle numbers of up to thirty-five decimal digits. These may well be interpreted as the significant figures in a number with an embedded decimal point, in which case accuracy to thirty-five places is by no means an extravagant requirement in many spheres of activity. Now, $\log_{10} 2 \approx 0.3$, so 2^{117} is of the same order of magnitude as 10^{35} . Thus we shall need about 117 bits to represent all numbers up to 10^{35} ; eight PDP-11 words, 128 bits, will be both necessary and sufficient.

The representation of multiple precision numbers is the same as that of numbers housed in a single word, except that the number of bits allotted is greater. Consider an eight-word number housed in the block `MEM` to `MEM + 16` (check the count!). If the number is 1, all bits in `MEM` to `MEM + 14` will be clear; in `MEM + 16` bit 0 will be set, and all other bits will be clear. If the number is 400000, bit 17 of the entire number will be the only bit set in the block; this is bit 1 of `MEM + 14`.

Negative numbers will be represented here in 128-bit twos complement form. Thus -1 appears as a 1 in all bits of the block.

There is no inherent reason why successively higher addresses in the block at `MEM` should house bits of lower significance. It is somewhat intuitive, in that the diagrammatic representation



places both increasing addresses and bits of decreasing significance in their natural left-to-right ordering. It also conforms with special instructions on some PDP-11 CPUs that perform operations on double precision numbers with the lower of the two addresses housing the more significant bits. Hence this convention should be followed.

Each multiple precision integer "variable" will exist in our program as a pointer to the block housing its eight word-length "digits." Suppose our program will be working with ten such eight-word numbers. We must reserve a sufficient block of locations and set up a pointer to it.

<code>MPBLK:</code>	<code>.BLKW</code>	<code>120</code>
<code>MPVAR:</code>	<code>.WORD</code>	<code>MPBLK</code>

To declare a symbol as representing a multiple precision number, we must associate it with an eight-word block within the space at `MPBLK`. We can write a macro `MP`, so that `VAR` can be so declared by the call

<code>MP</code>	<code>VAR</code>
-----------------	------------------

A suitable macro would be

```
.MACRO      MP      X
            ADD      #20,MPVAR
            MOV      MPVAR,X
            BR       .+2
X:          .WORD    0
.ENDM
```

(Why do we prefer not to use a dummy label in the branch instruction?) Observe that **VAR** is set up as a pointer to the word following its eight-word block. This allows efficient use of autodecrement addressing later on.

The best approach is to write macros for all the arithmetical operations we will want to perform. Thus, in order that

MPCLR VAR

should set **VAR** to zero, we need

```
.MACRO      MPCLR      X      ?L1
            MOV      X,R1
            MOV      #10,R0
L1:         CLR      -(R1)
            SOB      R0,L1
.ENDM
```

EXERCISE: Write a macro to support the call **MPMOV VARA,VARB**.

Consider now a macro **MPINC**. It is not enough just to increment the lowest-order word (highest address), because that word might have contained **177777**; in that case, we must *carry* a 1 into the next word. The **INC** instruction is unhelpful to us here, because the way it sets the condition codes is fashioned for a signed integer in its operand. Note that for us the sign bit is bit 15 of the highest-order word (lowest address); all the other words simply provide sixteen bits apiece toward the total representation of the number. The **ADD** instruction, however, is perfect for our requirements, since it sets the C bit if a 1 was carried out of the word. So we start our **MPINC** routine by adding 1 to the lowest-order word. Then we increment the next word if the C bit was set, not otherwise. The single-operand instruction **ADC** is designed for this task; it adds the contents of the C bit into the destination address.

Possibly adding in the C bit causes a further carry; **ADC** sets the condition codes just like **ADD**, so we can repeat the preceding process on the next word, and so on through to the highest-order word. With a little trickery, we now have

```
.MACRO      MPINC      X      ?L1
            MOV      X,R1
            MOV      #10,R0
            SEC
L1:         ADC      -(R1)
            SOB      R0,L1
.ENDM
```

- EXERCISES:**
- (i) The instruction **SBC** subtracts the contents of the C bit from its operand, setting the condition codes like **SUB**. Write the **MPDEC** macro.
 - (ii) Write macros **MPASL** and **MPASR** to perform correct multiplication and division by two. (*Hint*. Use the C bit as a staging post for bits being shifted between words.)
 - (iii) Incorporate in all the relevant macros an overflow check on the arithmetical correctness of the result.
 - *(iv) Write a routine to print out a multiple precision integer in octal notation.

Consider how we performed the **MPINC** routine. For each word, after adding 1, we checked (using the C bit) whether the result was too much for a single word to house; if so, we carried 1 to the next word, and so on. This is exactly like adding 1 to an integer expressed in positional notation; if the result of adding 1 in a column is too large for a single digit, we set that column to 0 and “carry” 1 into the next column to the left. Thus we may regard each of our multiple precision integers as an eight- “digit” number. The value of each digit is the contents of the respective word; this is a positional notation with base $2^{16} = 65536$!

Viewing multiple precision arithmetic in this way makes addition quite easy. If the result of addition in one “column” exceeds the base, there will be a “carry” of 1 into the next column. Thus, at each stage, we **ADC** to the destination for the addition before the **ADD** into that word. Just as with **MPINC**, we must take into account that adding in the carried 1 might itself cause a further carry, and so on; this requires a loop within the main loop of the routine. The entire macro is in Figure B.1. Note that if the V bit is set by an **ADC** or addition into the highest-order word, the final result will be arithmetically incorrect. It may then be necessary to rewrite the program, allowing a higher degree of precision (that is, more words per number).

- EXERCISES:**
- (i) Write the macro **MPSUB**.
 - (ii) Write the macros **MPTST** and **MPCMP** so that *any* following branch instruction will work as it should. Should you amend **MPADD** and **MPSUB** to take this consideration into account?

We may sometimes need to change a number from single to multiple precision form. A plain **MOV** to the lowest-order word and clearing the other words will not work, because the number might be negative. The Sign eXTend instruction **SXT*** will clear its destination operand if the N bit is clear and set it to -1 if the N bit is set. **SXT** has no effect on the N and C bits. Thus, to set the multiple precision integer **X** equal to the contents of location **MEM**,

```

                                MOV      X,R1
                                MOV      #7,R0
                                MOV      MEM,-(R1)
L1:                            SXT      -(R1)
                                SOB      R0,L1

```

Readers with some knowledge of computational mathematics can go on to write their own **MPMUL** and **MPDIV** macros. Division by ten is particularly necessary, in order to perform output. If this is the only division a program performs, it saves programming labor for the program to maintain a table of powers of ten. Suppose we want to output a positive number. For each power of ten, starting from the highest, the number of times it can be subtracted from our number before the result becomes negative gives the corresponding decimal digit. This process is very simple to encode and is fairly efficient in operation.

*Not available on some smaller CPUs.

```

.MACRO  MPADD    X,Y      ?L1,?L2
        MOV      X,R1
        MOV      Y,R2
        MOV      #10,R0
        CLC
L1:     MOV      R2,-(SP)
        MOV      R0,-(SP)
L2:     ADC      -(R2)
        SOB      R0,L2
        BVS      ERROR
        MOV      (SP)+,R0
        MOV      (SP)+,R2
        ADD      -(R1),-(R2)
        SOB      R0,L1
        BVS      ERROR
.ENDM

```

Figure B.1 A macro to perform multiple precision addition.

FLOATING POINT ARITHMETIC

Until now we have discussed only *fixed point* arithmetic; the point referred to is the *radix point*. In radix, or base, ten, the radix point is the familiar decimal point. Just as columns to the left of the decimal point indicate successive multiplications by ten, so columns to the right of the decimal point indicate successive divisions by ten. Thus, D 12.34 denotes $1 \times (\text{ten}) + 2 \times (\text{one}) + 3 \times (\text{one-tenth}) + 4 \times (\text{one-tenth of one-tenth})$.

The same notation will serve in any base, with the value of the base replacing ten everywhere in the above description. Thus, O 12.34 denotes $1 \times (\text{eight}) + 2 \times (\text{one}) + 3 \times (\text{one-eighth}) + 4 \times (\text{one-eighth of one-eighth}) = 8 + 2 + 3/8 + 4/64$ in decimal notation. Here we are using an *octal* point.

Likewise, we may use a *binary* point. For example,

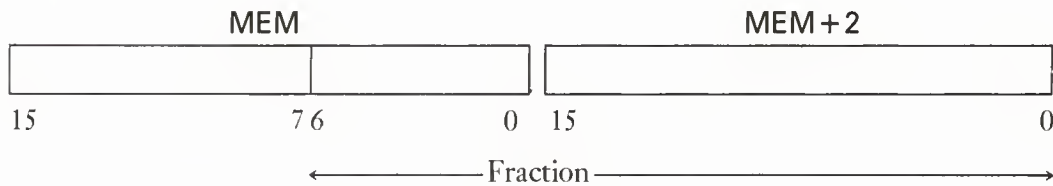
$$\begin{aligned}
 B\ 0.\dot{1}0 &= B\ 0.101010 \dots = D\ 1/2 + 1/8 + 1/32 + \dots \\
 &= D\ 2/3
 \end{aligned}$$

(sum the geometric series).

EXERCISE: Determine the binary expansion of D 1/10. Use it to develop another method of performing division by ten.

The radix point here is *fixed* in the sense that, if there is one, the programmer knows exactly where it is assumed to be. Multiplication of D 123456 by D 234567 could be handled by the same machine instructions as multiplication of D 12345600 by D 0.0234567. The machine is, in fact, multiplying the integers comprising the significant figures of the operands. Elsewhere the program would keep a record of the position of the radix point. In the preceding example we have a decimal point. If shift instructions are used to reposition numbers within computer words, the position of a binary point is affected.

MACRO-11 has provisions for handling numbers in a special *floating point format* that is compatible with hardware floating point processors available as an optional accessory with most CPUs. The PDP-11 floating point representation of a number is in three parts. At bit 6 in the first word of the representation the binary fractional part begins. This extends down to bit 0, then through as many more words as the representation consists of. The total number of words may be one, two, or four. Thus, with two-word representation, we have



The fractional part of a floating point number is related to the exponent. Thus, in decimal notation, we may represent 123.45 as 0.12345×10^3 , or as 0.012345×10^4 , and so on indefinitely. The first form, characterized by having the first numeral after the radix point not zero, is called the *normalized* floating point representation of the number. PDP-11 processors work exclusively with normalized binary floating point representation. In this form the first digit following the binary point must be 1; *this digit is not stored in the floating point representation* (it is called the “hidden bit”). You can imagine a binary 0.1 preceding bit 6 of the first word.

Decimal floating point numbers may be passed to the assembler in the form of a string of decimal digits, including a decimal point if necessary. A scale factor may be provided by appending the letter **E** followed by a decimal integer (which may be negative or zero). The number is specified in a **.FLT2** (two-word representation) or **.FLT4** (four-word representation) directive. Thus

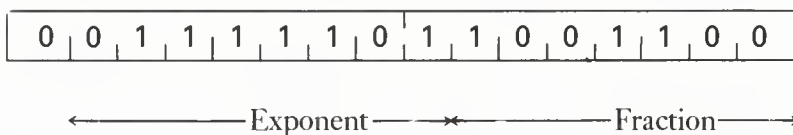
MEM: .FLT2 -2.1E-3

will assemble the normalized *binary* floating point representation of $D -0.0021$ in two words starting at **MEM**.

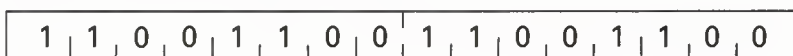
Bit 0 of the first word gives the sign in the usual way. Bits 7 to 14 represent the binary exponent. For positive floating point numbers, the actual quantity housed in these bits is **O 200** more than the exponent. Consider, for example, $D 1/10$. It is easy to show that

$$\begin{aligned}
 D 1/10 &= 3/32 (1 + 1/16 + 1/64 + \dots) \\
 &= B 0.0001100110011 \dots \\
 &= 2^{-3} \times B 0.110011001100 \dots
 \end{aligned}$$

in normalized binary form. So the exponent field will contain $O 200 - 3 = 175 = B 01111101$. Thus, remembering to suppress the leading digit of the fractional part, the first word is



or **037314**. The second word is



or **146314**. The last octal digit of the second word would, in fact, be rounded up to **5** by the assembler.

The floating point representation of zero in the PDP-11 is anything smaller in magnitude than $D 2^{-127}$; that is, anything with the exponent field zero, representing an exponent of -128 .

Clearly, arithmetical operations on floating point numbers must be carried out by instructions that take the floating point format specifically into account. Consider, for example, how the sum $D 27 + 3$ is formed when these quantities are stored as floating point numbers. Their normalized binary floating point representations are, using octal notation for the fractional parts: $D 27 = O 33 = 2^5$

$\times 0.66$; $D\ 3 = 2^2 \times 0.6$. First, the representation of the number with smaller exponent is amended to make the exponents match. In this case, the result is to represent $D\ 3$ as $2^5 \times 0.06$. Of course, this is no longer a normalized representation. Now the addition is performed:

$$2^5 \times 0.66 + 2^5 \times 0.06 = 2^5 \times 0.74$$

Both of these numbers can be represented exactly in single-word floating point format, as can their sum. This form is achieved with the numeric control assembler operator $\wedge F$ (up arrow, then F), which causes the number following to be interpreted as decimal floating point.

```
MOV      ^F27, MEM
```

A routine to perform the preceding floating point addition might begin by shifting the fractional part of the representation of $D\ 3$ three bits to the right, to make the exponents match. The “hidden” leading digit must, of course, be brought in. Adding the fractional parts would then give the correct result, already normalized.

In general, however, this method is not adequate. Consider $D\ 27 + 6$, with both numbers represented in single-word format. $D\ 27$ has exponent field **O 205**. Its binary fractional expansion is 0.11011000, to eight bits; omitting the leading digit, we get a 7-bit fraction field equal to **O 130**. $D\ 6$ has exponent field **O 203** and fraction field **O 100** in the normalized form. Shifting this two bits to match exponents with $D\ 27$ gives a fraction field equal to **O 60** (remembering the hidden bit). Addition of the fractional parts now gives **O 210**, which overlaps into the exponent field.

Such calculations are best left to the hardware *floating point processor*, if your CPU is equipped with one; its operations are also considerably faster than software routines. We can do no more here than briefly illustrate the use of the processor.

The floating point processor operates only with two- or four-word-length *normalized* numbers, always giving its results in the same form. Operations on these numbers are performed by special assembler language instructions that are automatically referred to the processor. Thus the ADD Floating instruction **ADDF** performs addition of two-word-length numbers, while the ADD Double instruction **ADDD** is used for four-word-length numbers.

Floating addition can only be carried out if one of the numbers is housed in one of four *accumulators* within the processor. (There are more than four processor accumulators, but the rest cannot be used for data transfers between memory and the processor.) These accumulators may be addressed by the same symbols as the first four CPU registers; the hardware ensures that the floating point processor locations are actually referenced. To avoid confusion, however, it is best to declare

```
F0 = %0
```

and so on; the symbols **AC0**, and so on, are also commonly used.

Let us proceed with the calculation that proved too difficult previously. First, we set up the floating point representations.

```
MEM:      .FLT2      27
          .FLT2      6                ;at MEM+4
```

Now we must load one of the numbers into an accumulator (its length will be adequate), with the Load Floating instruction

```
LDF      MEM, F0
```

Addition always forms its result in the accumulator.

```
ADDF      MEM+4,F0
```

Finally, we can STore the result back into memory.

```
STF      F0,WRD
```

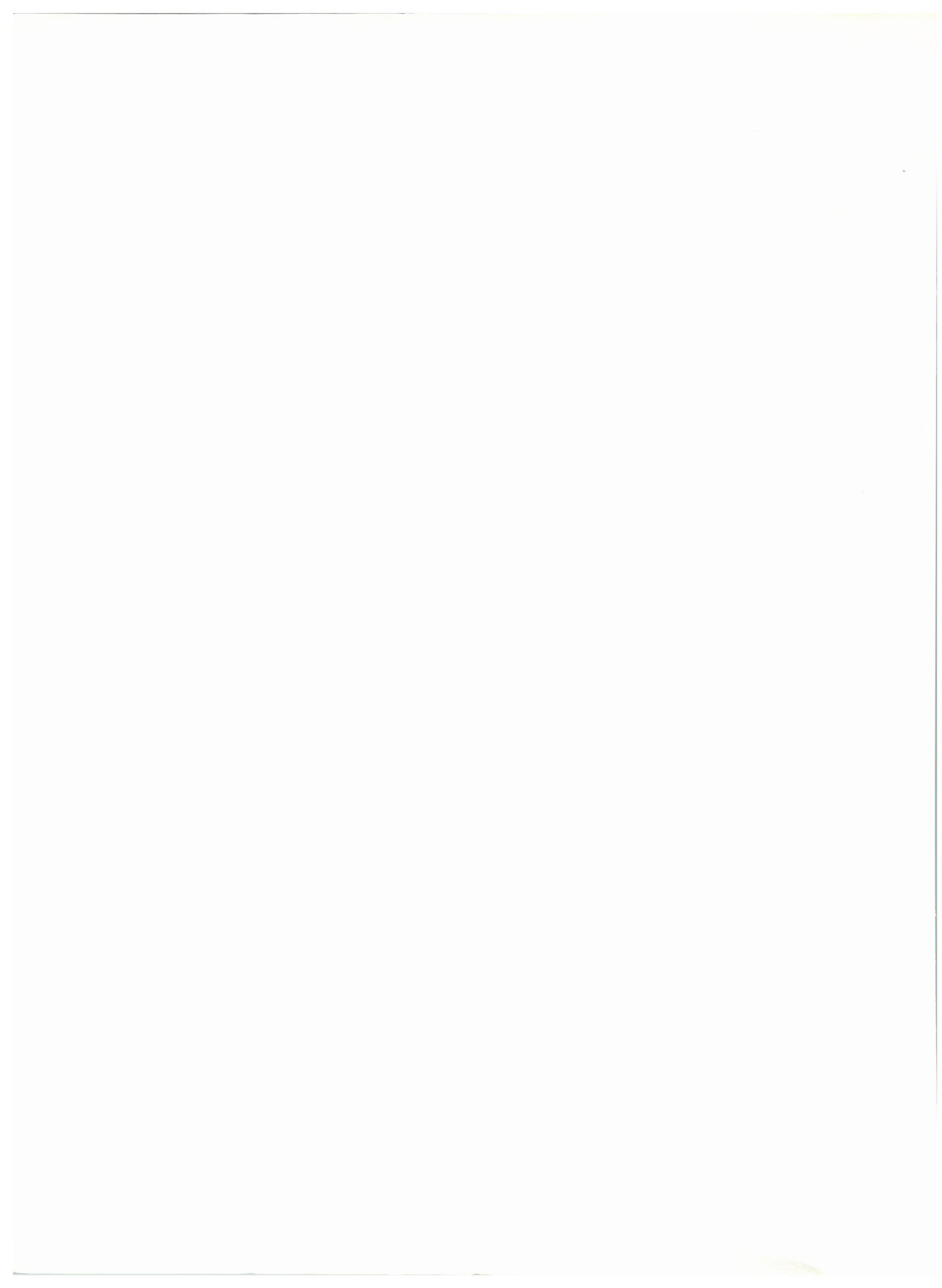
At one time, all computers required calculations to be performed via an accumulator in this way; quite a few still do. The floating point processor thus gives us a nostalgic glimpse of a fortunately waning tradition.

ASCII Codes

CHARACTER	CODE	CHARACTER	CODE
NULL	0	@	100
CONTROL-A	1	A	101
CONTROL-B	2	B	102
CONTROL-C	3	C	103
CONTROL-D	4	D	104
CONTROL-E	5	E	105
CONTROL-F	6	F	106
BELL	7	G	107
BACKSPACE	10	H	110
TAB	11	I	111
LINE FEED	12	J	112
VERT TAB	13	K	113
FORM FEED	14	L	114
CARR RETN	15	M	115
CONTROL-N	16	N	116
CONTROL-O	17	O	117
CONTROL-P	20	P	120
CONTROL-Q	21	Q	121
CONTROL-R	22	R	122
CONTROL-S	23	S	123
CONTROL-T	24	T	124
CONTROL-U	25	U	125
CONTROL-V	26	V	126
CONTROL-W	27	W	127
CONTROL-X	30	X	130
CONTROL-Y	31	Y	131
CONTROL-Z	32	Z	132
ESCAPE	33	[	133
CONTROL-\	34	\	134
CONTROL-]	35	]	135
CONTROL-^	36	^	136
CONTROL- ₋	37	₋	137

CHARACTER	CODE	CHARACTER	CODE
SPACE	40		140
!	41	a	141
"	42	b	142
#	43	c	143
\$	44	d	144
%	45	e	145
&	46	f	146
'	47	g	147
(	50	h	150
)	51	i	151
*	52	j	152
+	53	k	153
,	54	l	154
-	55	m	155
.	56	n	156
/	57	o	157
0	60	p	160
1	61	q	161
2	62	r	162
3	63	s	163
4	64	t	164
5	65	u	165
6	66	v	166
7	67	w	167
8	70	x	170
9	71	y	171
:	72	z	172
;	73	{	173
<	74		174
=	75	}	175
>	76	~	176
?	77	RUBOUT	177

Index of Macro-11 Instructions



Period as first symbol indicates a pseudo-op.

ADC(B), 163

ADD, 18

.ASCII, .ASCIZ, 65, 67

.ASECT, .CSECT, 78

ASL(B), ASR(B), 87

BIC(B), BIS(B), BIT(B), 80, 81

.BLKB, .BLKW, 43, 78

BPT, 100

Branch instructions, 32, 83, 85

Clear condition code instructions, 85

CLR(B), 29

CMP(B), 35, 69, 86

COM(B), 81

DEC(B), 33

DIV, 29, 161

EMT, 98

.ENABL, .DSABL, 98

.END, 20

.ENDC, 106

.ENDM, 102

.EVEN, 68

.GLOBL, 154

HALT, 40

.IF, .IFF, .IIF, 106

INC, 17

.IRP, .IRPC, 133, 134

JMP, 99

JSR, 60

.LIST, .NLIST, 23, 133

.MACRO, 102

.MCALL, 21

MFPI, MTPI, 151

MOV(B), 15, 66, 86

MUL, 29, 161

NEG(B), 35

.NTYPE, 109

.RADIX, 106

.RAD50, 129

RESET, 143

ROL(B), ROR(B), 88

RTI, 97

RTS, 63

RTT, 126

SBC(B), 164

Set condition code instructions, 85

SOB, 89

SPL, MTPS, 124

SUB, 18

SWAB, 87

SXT, 164

.TITLE, 23

TRAP, 99

TST(B), 36

WAIT, 126

.WORD, .BYTE, 40, 67

XOR, 81

Subject Index

- Accumulator, 167
- Address, 15
 - absolute/relocatable, 24, 44, 55, 78
 - base, 146
 - effective, 49
 - physical/virtual, 144
 - relative, 23, 55
 - start, 20
- Addressing modes, 49, 97
- ASCII, code, 13
 - text, 65
- Assembler, 2, 17
 - error messages, 53, 111
- Base, 11
- Binary, notation, 11
 - point, 165
- Bit, 11
- Block, 132, 147
 - structure, 58
- Bootstrap, 116
- Branch, 33, 58
- Breakpoint (ODT), 158
- Buffer, editor, 5, 10
 - memory, 27, 124, 134
 - peripheral devices, 18, 112, 115, 134
- Bug, 44
- Bus, 119
- Byte, 15, 66, 86
- CARRIAGE RETURN, 9
- Carry, 84
- C bit, 84, 164
- Central Processing Unit (CPU), 18, 52
 - modes, 149
- Channel, 130
- Clock, 126
- Comments, 32
- Compile, 22
- Complementary function, 35
- Conditional assembly, 106
- Condition codes, 37, 83, 84, 85
- Console, 113, 151
- CONTROL characters, 1
- Core, 5, 11, 40
- Coroutines, 76, 124, 152
- Debugging, 43, 100, 153
- Decimal point, 165
- Delimiter, 65
- Destination, 25, 51, 89
- Device handler, 129
- Direct assignment, 25, 78
- Directory, 6
- Disk, 1, 129, 136
- Dispatch table, 99
- Documentation, 3
- Done bit, 115
- Double precision, 162
- Dummy argument, 102
- Echo, 2, 27, 116
- Editor, 3
- Effective address, 49
- Enable bit, 117, 141, 145
- ESCAPE, 5
- Execution cycle, 52
- File, 3, 22
 - I/O to and from, 5, 130
- File name, 4, 24
 - extensions, 5, 24
- Fixed point, 165
- Flip-flop, 11
- Floating point, 165
- Flowchart, 39, 47, 63, 74, 77
- Forward reference, 110
- Fraction, 166
- Global symbols, 154
- Hardware, 1
- Hidden bit, 166
- Higher level languages, 92
- High order word, 161
- Illegal references, 99
- Immediate mode addressing, 54
- Index, mode, 49
 - register, 41
- Indirect addressing, 50
- Input/Output (I/O), 2, 11, 18, 122, 130, 139
- Instruction, encoding, 49, 89
 - execution, 52
 - interpretation, 77
 - representation in memory, 18
- Instructions, arithmetical/logical, 79
 - byte/word oriented, 67
 - single/double operand, 48, 51
- Interface, 120
- Interrupt, 118, 136, 141
- Job, 4
- Keyboard, 115, 121
- Label, 20
 - local, 69, 104

- Last in, first out, 69
- Library, 22, 127
- Line (editor), 7
- Linked list, 52
- Linker, 21, 153
- Literal, 93
- Loader, 2, 20
- Local symbol, 69
- Location counter, 77
- Logical, error, 53
 - operations, 85
- Loop, 34
- Low order word, 161

- Macros, 20, 101, 163
- MACRO systems program, 20
- Memory, 5, 40
 - address, 15
 - page, 144
 - protection, 148
 - read only (ROM), 113
- Module, 90, 154
- Monitor, 2
 - calls, 127
 - intervention, 119
- Multiple precision, 162

- N bit, 37
- Negative numbers, 35, 56, 86
- Nesting, conditional directives, 107
 - interrupts, 125
 - macros, 105
 - subroutines, 62
- No-op, 37, 85
- Normalized form, floating point, 166

- Operating system, 2
- Output, *see* Input/Output
- Overflow, 82

- Page, editor, 9
 - memory, 144
- Paper tape, 117
- Parameters, 90
- Parity, 87
- Password, 4
- Patching, 100
- Peripheral devices, 1, 129
- Pointer, 15, 42, 50, 162
 - editor, 7
 - linkage, 62
 - ODT, 155
 - stack, 69
 - stepping, 45
- Position independent code (PIC), 56, 98, 113
- Printer, 19, 113
- Priority, device interrupt, 119
- Processor status word (PS), 37
- Program, 3, 17
 - counter (PC), 17, 54
 - design, 58
 - listing, 22, 133, 154
 - self-modifying, 117

- Pseudo-op, 21
- Pushdown list, 69

- Radix, 11
- Radix-50 code, 128
- Read, from memory, 11
 - from peripheral devices, 135, 139
- Read only bit, 115
- Ready bit, 115
- Recursion, 73
- Register, 25, 115, 137, 154
 - assignment statement, 25
 - index, 41
 - instruction, 18
 - linkage, 62, 76
- Relocation, 24, 144
 - constant, 78
 - registers (ODT), 154
- Repeat directives, 133
- Return, from interrupt, 97, 126
 - from subroutine, 63
- RT-11 operating system, 19, 127

- Sign bit, 56, 86
- Software, 2
- Sorting, 46, 122
- Source, 25, 51
- Stack, 64, 69
 - pointer, 69
 - system, 74
 - user/kernel, 151
- Standalone system, 1
- Stored program, 16
- Subroutines, 58
 - calling, 58, 93
 - linkage, 62
 - nesting, 62
 - parameters, 90
 - recursive, 73
- Symbol table, 40
- Syntax, 5, 102
 - error, 53
- Systems programs, 2

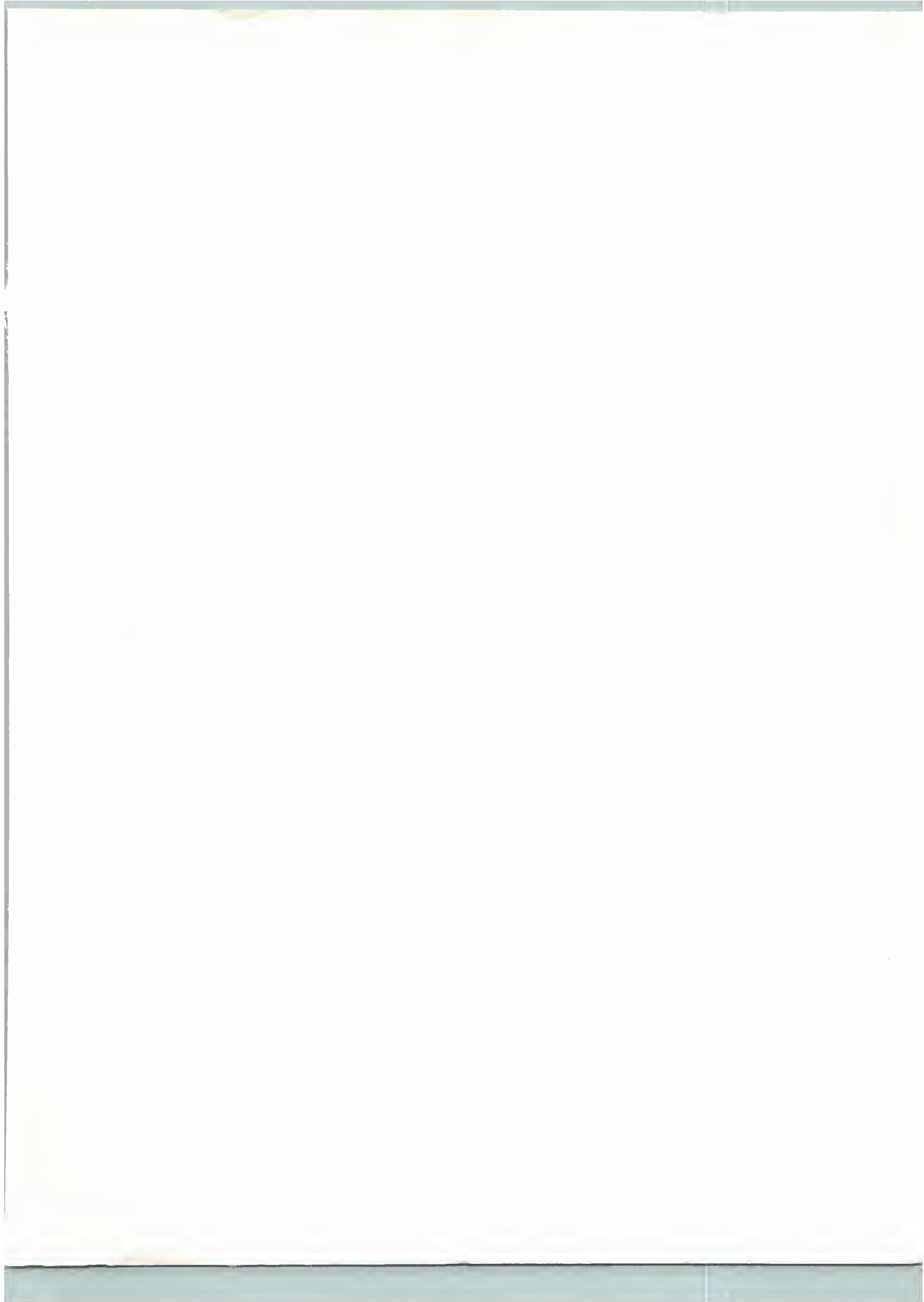
- Tape, 1
- T bit, 100, 127
- Terminal, 1, 112, 122
- Timesharing system, 1
- Tracing, 44
- Traps, 94, 98
- Twos complement, 56

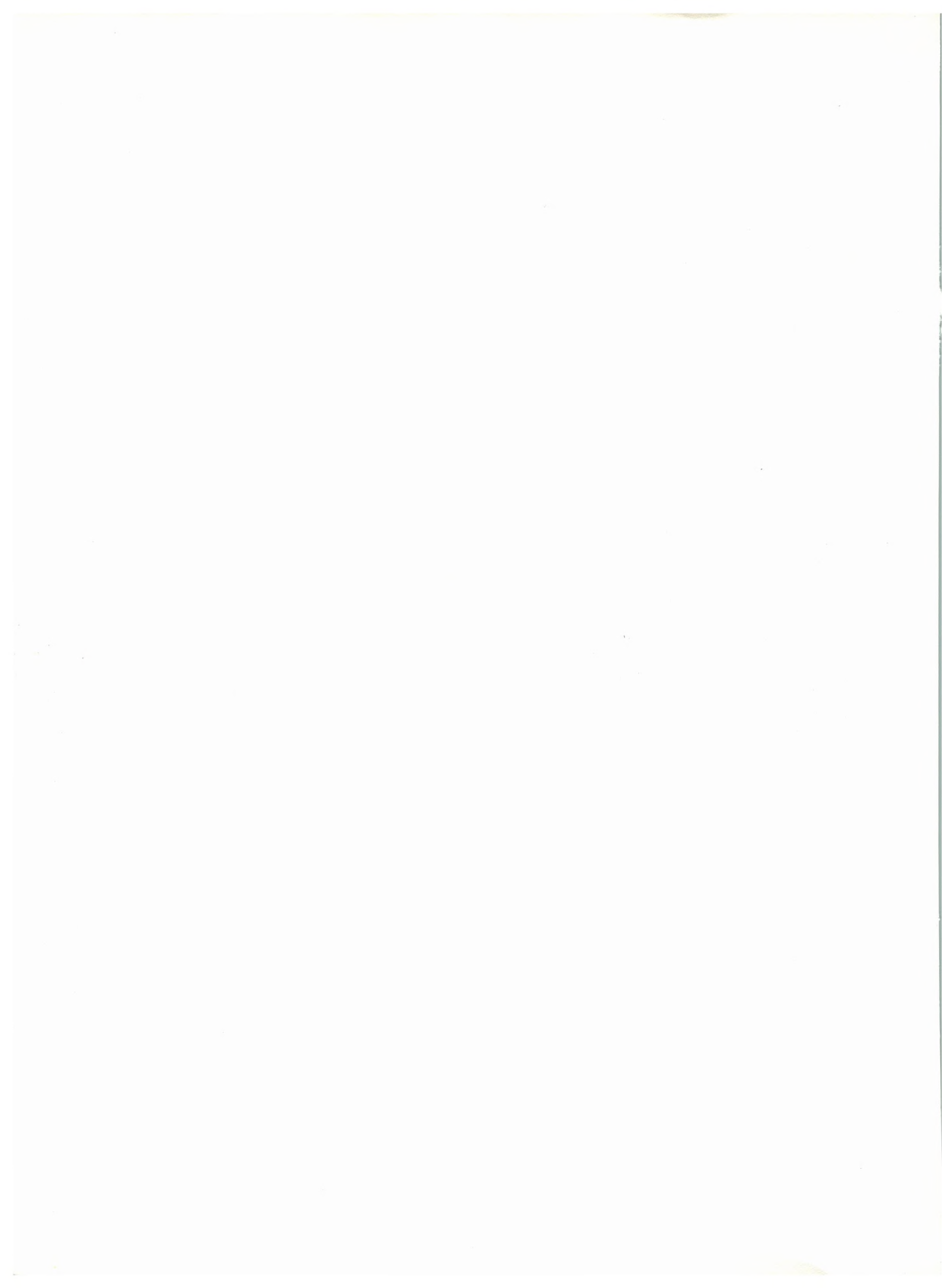
- UNIBUS, 119, 143

- V bit, 83
- Vector, 95, 98, 120, 148

- Word, 11
 - interpretation of contents, 13, 18
- Write, to memory, 11
 - to peripheral devices, 132, 139

- Z bit, 37





	<i>Mnemonic</i>	<i>Code</i>	<i>Description</i>	<i>Condition Codes Affected</i>
DOUBLE OPERAND	MOV(B)	x1SSDD	move src to dst	N,Z set/cclr by result, V clr
	ADD	06SSDD	add src to dst	all set/cclr by result
	SUB	16SSDD	subtract src from dst	all set/cclr by result
	CMP(B)	x2SSDD	form (src - dst)	all set/cclr by result
	BIS(B)	x5SSDD	put (src OR dst) in dst	N,Z set/cclr by result, V clr
	BIT(B)	x3SSDD	form (src AND dst)	N,Z set/cclr by result, V clr
	BIC(B)	x4SSDD	put (~ src AND dst) in dst	N,Z set/cclr by result, V clr
	MUL	070RSS	multiply } result in R, and	all set/cclr by result
	DIV	071RSS	divide } next reg if R odd	all set/cclr by result
	XOR	074RDD	exclusive OR, result in dst	N,Z set/cclr by result, V clr
PROGRAM CONTROL	JSR	004RDD	reg → stack, PC → reg, dst → PC	
	RTS	00020R	reg → PC, stack → reg	
	SPL	00023L	set CPU priority level to L	
	JMP	0001DD	dst → PC	
	SOB	077RXX	XX = offset; subtract 1 from reg contents, if ≠ 0 branch back	
	EMT	104000 - 104377	PS,PC → stack, new PC,PS from 30,32	all loaded from 32
	TRAP	104400 - 104777	PS,PC → stack, new PC,PS from 34,36	all loaded from 36
	BPT	000003	PS,PC → stack, new PC,PS from 14,16	all loaded from 16
	RTI	000002	load PC,PS from stack	all loaded from stack
	RTT	000006	PC,PS from stack, delay T trap	all loaded from stack
CPU	MFPI	0065SS	move word from previous space to current stack	N,Z set/cclr by src value, V clr
	MTPI	0066DD	move word from current stack to previous space	N,Z set/cclr by result V clr
	HALT	000000	stop CPU	
	WAIT	000001	wait for interrupt	
	RESET	000005	reset UNIBUS	
CODE OPS	CLC	000241	clear C bit	
	CLV	000242	clear V bit	
	CLZ	000244	clear Z bit	
	CLN	000250	clear N bit	
	CCC	000257	clear all codes	
	<i>Mnemonic</i>	<i>Code</i>	<i>Description</i>	<i>Mnemonic</i> <i>Code</i> <i>Description</i>
	SEC	000261	set C bit	
	SEV	000262	set V bit	
	SEZ	000264	set Z bit	
	SEN	000270	set N bit	
	SCC	000277	set all codes	



ISBN 0-471-04905-0